

**Programmer's
Reference Manual**

1985

3803



RC 3803 CPU Programmer's Reference Manual

Author: Knud Henningsen
Technical Editor: Knud Erik Hansen

KEY WORDS: RC3803, CPU 720, Revision 0.

ABSTRACT: This paper describes the logical structure of the RC3803 Central Processor Unit.

Users of this manual are cautioned that the specifications contained herein are subject to change by RC at any time without prior notice. RC is not responsible for typographical or arithmetic errors which may appear in this manual and shall not be responsible for any damages caused by reliance on any of the materials presented.

Copyright © A/S Regnecentralen af 1979
Printed by A/S Regnecentralen af 1979, Copenhagen

Table of contents

1.	RC3803 SPECIFICATIONS	1
1.1	Central Processor Unit	1
1.2	Memory	1
1.3	Input/Output	2
1.4	Interrupt Capability	2
1.5	Data Channel	3
1.6	Power Fail/Auto Restart	3
1.7	Real Time Clock	3
1.8	Diagnostic Front Panel	4
2.	INTERNAL CONFIGURATION	5
2.1	Introduction	5
2.2	Program Structure	5
2.2.1	Program Execution	5
2.2.2	Program Flow Alteration	6
2.2.3	Program Size	8
2.2.4	Program Flow Interruption	8
2.3	Information Formats	9
2.3.1	Fundamental Concepts	10
2.3.2	Bit Numbering	10
2.3.3	Binary Representation	11
2.3.4	Octal Representation	13
2.3.5	Hexadecimal Notation	14
2.4	Numerical Quantities	14
2.4.1	Integers	14
2.4.2	Logical Quantities	17
2.5	Addressing	17
2.5.1	Word Addressing	17
2.5.2	Byte Addressing	21
3.	INSTRUCTIONS	23
3.1	Introduction	23
3.2	Instruction Formats	23
3.3	Mnemonic Description	23

<u>TABLE OF CONTENTS (continued)</u>	<u>PAGE</u>
3.4 Program Flow Control	24
3.4.1 JUMP	25
3.4.2 JUMP TO SUBROUTINE	26
3.4.3 INCREMENT AND SKIP IF ZERO	27
3.4.4 DECREMENT AND SKIP IF ZERO	27
3.5 Data Transfer Operation	28
3.5.1 LOAD ACCUMULATOR	29
3.5.2 STORE ACCUMULATOR	29
3.6 Integer Arithmetic and Logical Operations	30
3.6.1 ADD	36
3.6.2 SUBTRACT	37
3.6.3 NEGATE	38
3.6.4 ADD COMPLEMENT	39
3.6.5 MOVE	39
3.6.6 INCREMENT	40
3.6.7 COMPLEMENT	40
3.6.8 AND	41
3.6.9 Examples	41
4. INPUT/OUTPUT	45
4.1 Introduction	45
4.2 Operation of I/O Devices	46
4.3 Interrupt System	46
4.4 Priority Interrupts	48
4.5 Direct Memory Access Data Channel	50
4.6 I/O Instructions	51
4.6.1 DATA IN A	52
4.6.2 DATA IN B	53
4.6.3 DATA IN C	53
4.6.4 DATA OUT A	54
4.6.5 DATA OUT B	54
4.6.6 DATA OUT C	55
4.6.7 I/O SKIP	55
4.6.8 NO I/O TRANSFER	56

<u>TABLE OF CONTENTS (continued)</u>	<u>PAGE</u>
4.7 Central Processor Functions	56
4.7.1 INTERRUPT ENABLE	58
4.7.2 INTERRUPT DISABLE	58
4.7.3 READ SWITCHES	59
4.7.4 INTERRUPT ACKNOWLEDGE	59
4.7.5 MASK OUT	60
4.7.6 I/O RESET	60
4.7.7 HALT	61
4.7.8 CPU SKIP	61
5. PROCESSOR FEATURES	62
5.1 Introduction	62
5.2 Power Fail	63
5.3 MEMORY EXTENSION	64
5.4 CPU IDENTIFY	65
5.5 Byte Manipulation	65
5.5.1 LOAD BYTE	66
5.5.2 STORE BYTE	67
5.6 BYTE MOVE	68
5.7 WORD MOVE	69
5.8 SEARCH ITEM	70
5.9 SEARCH FREE	73
5.10 PROCESS LINK	74
5.11 PROCESS REMOVE	76
5.12 PROCESS LINK PRIORITY	78
5.13 INSTRUCTION FETCH (MUSIL)	81
5.14 TAKE ADDRESS (MUSIL)	82
5.15 TAKEVALUE (MUSIL)	84
5.16 COMPARE Byte Strings	86
6. PROCESSOR OPTIONS	88
6.1 Real Time Clock	88
6.2 Teletype Controller	89
6.2.1 Instructions	89
6.2.2 Programming	92
6.2.3 Programming Examples	93

<u>TABLE OF CONTENTS (continued)</u>	<u>PAGE</u>
7. PROGRAM LOADING	97
7.1 Introduction	97
7.2 Automatic Loading	98
8. SWITCHES AND INDICATORS	103
8.1 Switches	103
8.1.1 ENABLE TCP	103
8.1.2 AUTOLOAD DEVICE SELECT	104
8.1.3 PARITY ERROR	104
8.1.4 MEMORY EXTENSION SELECT	105
8.2 Indicators	106
8.2.1 PARITY ERROR	106
8.2.2 CPU-STATUS	106

APPENDICES:

A. I/O DEVICE CODES AND MNEMONIC	108
B. ASCII CHARACTER CODES	111
C. DOUBLE PRECISION ARITHMETIC	117
D. INSTRUCTION USE, EXAMPLES	119
E. INSTRUCTION EXECUTION TIMES	126

1. RC3803 Specifications 1.

1.1. Central Processor Unit 1.1

The RC3803 Central Processor Unit is a micro-programmed, general purpose stored-program computer with four accumulators. The CPU works on the basis of a unit of information called a word which consists of 16 bits. Arithmetic and logical operations are performed on operands held in the accumulators, which consequently also are 16 bits in length. Two of the accumulators can be used as index registers for addressing purposes.

1.2 Memory 1.2

The main memory is available in two alternative modules:

RC3608 is a core memory with a capacity of 32K words and a cycle time of 750 ns.

RC3609 is a core memory with a capacity of 16K words and a cycle time of 650 ns.

The CPU can directly address 32K words of core memory and provides for base page, relative, indexed and multi-level indirect addressing modes. By the use of a special instruction the CPU can be switched to a mode which will allow it to work with up to 64K words of core memory.

Word length in memory is $16 + 2 = 18$ bits. The two extra bits are parity check bits. They are generated during each memory write cycle and are checked during each memory read cycle. The detection of a parity error can affect the operation of the CPU in two alternative ways: the error can be indicated on the front frame of the CPU board while processing continues uninterrupted or processing can be brought to a halt. The selection of either possibility is left to the operator's choice by means of a switch also located on the CPU frame.

1.3 Input/Output

1.3

All peripheral devices are connected to the CPU through the Input/Output bus. This consists of a six-line device selection network, interrupt circuitry, command circuitry, and sixteen data transmission lines. Each individual Input/Output device has a unique six-bit device code and will only respond to commands if its own device code is transmitted through the device selection network of the Input/Output bus.

The six bits in the device code allows for 64 separate codes. A number of these codes are reserved for specific uses, but the remaining codes make it possible to obtain an extremely flexible handling of Input/Output devices.

1.4 Interrupt Capability

1.4

The interrupt circuitry included in the Input/Output bus provides the capability for any peripheral device to interrupt normal program execution whenever the device is in need of attention. When a peripheral device has requested an interrupt the processor will transfer control of operations to the main interrupt service routine, which will handle the servicing of the device. The interrupt service routine will establish the source of the interrupt either by polling all Input/Output devices connected to the CPU or it can use a special instruction to identify the device in question.

The interrupt system also provides the capability of implementing up to sixteen levels of priority in connection with interrupts, so that each individual peripheral device is associated with a specific priority level. A standard priority assignment is implemented by Regnecentralen, but the programmer can change these assignments according to his own choice.

1.5 Data Channel

1.5

Data transfers between peripheral devices and main memory under program control occupy processor time and retard the rate of information transfer.

To avoid this restriction the Input/Output bus contains circuitry allowing high-speed access direct to memory through the data channel, this permits a peripheral device to transfer data directly into/out of memory using a minimum of processor time. At the maximum transfer rate the data channel effectively stops the processor, but at lower rates processing continues while the data transfer takes place.

1.6 Power Fail/Auto Restart

1.6

The RC3803 computer incorporates a feature providing for automatic restart in the event of an unexpected power loss. The delay between the initial decrease of voltage and the actual automatic shut-down of the processor is utilized to bring the interrupt service routine into action. This routine will under these circumstances use the available interval of time to store the contents of accumulators, the program restart address and other information that will be necessary for restart and continued operation when the power supply again has been restored.

The Power Fail feature is entirely automatic and will restart operations on its own whenever power is again available.

1.7 Real Time Clock

1.7

A Real Time Clock can optionally be included in the RC3803 computer. This clock will generate a train of pulses independently of processor timing, this will allow the interrupt system to be activated at precisely spaced intervals of time. The pulse train frequency can be selected by the programmer among the following four possibilities: 10 Hz, 50 Hz, 100 Hz, and 1000 Hz.

1.8 Diagnostic Front Panel

1.8

A Diagnostic Front Panel can be connected to the CPU even during program execution. This will allow external, manual control of the CPU and will thus facilitate error detection and correction. The Diagnostic Front Panel is not described in detail in this manual, for further information concerning this consult the Reference Manual for the Diagnostic Front Panel - RCSL 52-AA542.

2. Internal Configuration 2.

2.1 Introduction 2.1

This chapter and the following deal in some detail with the basic concepts underlying the actual modus operandi of the RC3803 CPU. A more intimate knowledge of this subject is not strictly necessary for ordinary everyday use of the computer, because the high-level programming languages available are designed to allow symbolic programs to be written without reference to the more specific information contained in this manual. Thus the intention is not to establish guidelines for actual programming, for which purpose separate manuals are available, but to provide a source of background information for the programmer and/or operator.

2.2 Program Structure 2.2

Information about the type of operation - arithmetical or other - which the computer at any particular time must perform, is given to the CPU in the shape of an "instruction". The CPU will carry out successive instructions in strict sequence according to the order in which the instructions have been specified. The complete set of instructions is called a "program" and this must at the time of execution reside in main memory in order to be accessible to the CPU.

2.2.1 Program Execution 2.2.1

Each individual instruction occupies a space in memory called a "word" and although these words will usually occupy adjacent physical locations in memory, the program may incorporate instructions with the specific purpose of altering the sequence in which the instructions should be carried out.

Thus the CPU must be able to locate the correct word at the correct point in the sequence in order to execute the program properly. The actual physical location of a word is called its "address" and consequently the establishing of location is called "addressing".

Addressing the instructions is arranged by incorporating a counting circuit called the "program counter". The program counter contains one integer number, which always indicates the memory address of the instruction currently being carried out. When the operation specified by that particular instruction has been completed, the number in the program counter is incremented by one and the CPU will then retrieve the next instruction to be carried out from the memory location now being indicated by the number in the program counter. Succeeding addresses will thus form a strictly ascending numerical sequence and this method of operation is consequently called "sequential operation".

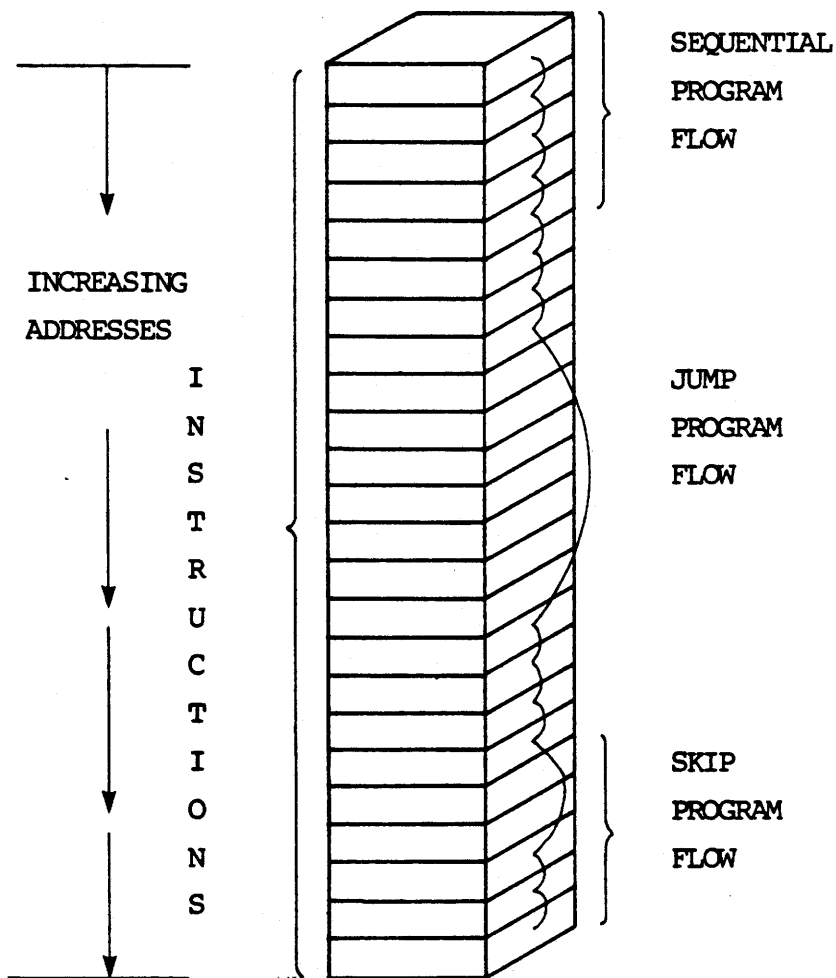
2.2.2 Program Flow Alteration

2.2.2

The programmer can however purposely arrange to deviate from the strict sequential operation. This is done by using the appropriate program flow control instructions which will make it possible to achieve two distinctly different types of program flow variation.

The "jump" type instruction will cause an arbitrary new number - either larger or smaller than the current one - to be inserted in the program counter. Thus when the jump instruction has been executed, the next instruction to be located can have any of all the possible addresses.

The "conditional skip" type instruction will first determine whether a specified test condition is true or not. If true, it will then cause the program counter to be increased by one, if false, nothing further will be done. When the conditional skip instruction has been executed, the program counter will be increased by one as in the usual sequential operation and thus the next instruction to be located will have either of the two following addresses depending on the outcome of the test. Normal sequential operation will be resumed after the completion of either type of instruction - using the updated value of the program counter - and will continue until the next program flow alteration occurs. An illustration showing the two types of program flow alteration appears in fig. 2.2.2.



Figur 2.2.2

2.2.3 Program Size

2.2.3

The integer number contained in the program counter will have a magnitude between 0 and 32,767 (both included) and will thus make it possible to address 32,768 separate memory locations which is then the maximum program size. The program need not necessarily start in memory location 0, but if the program counter reaches the value 32,767 the next incrementation will produce the value 0 and sequential operation will then continue from here as previously explained. Notice should be taken of the fact, that no indication whatsoever of this particular situation will be given.

NOTE: The proceeding outlined above will change if Memory Extension has been selected (cf. Section 5.3).

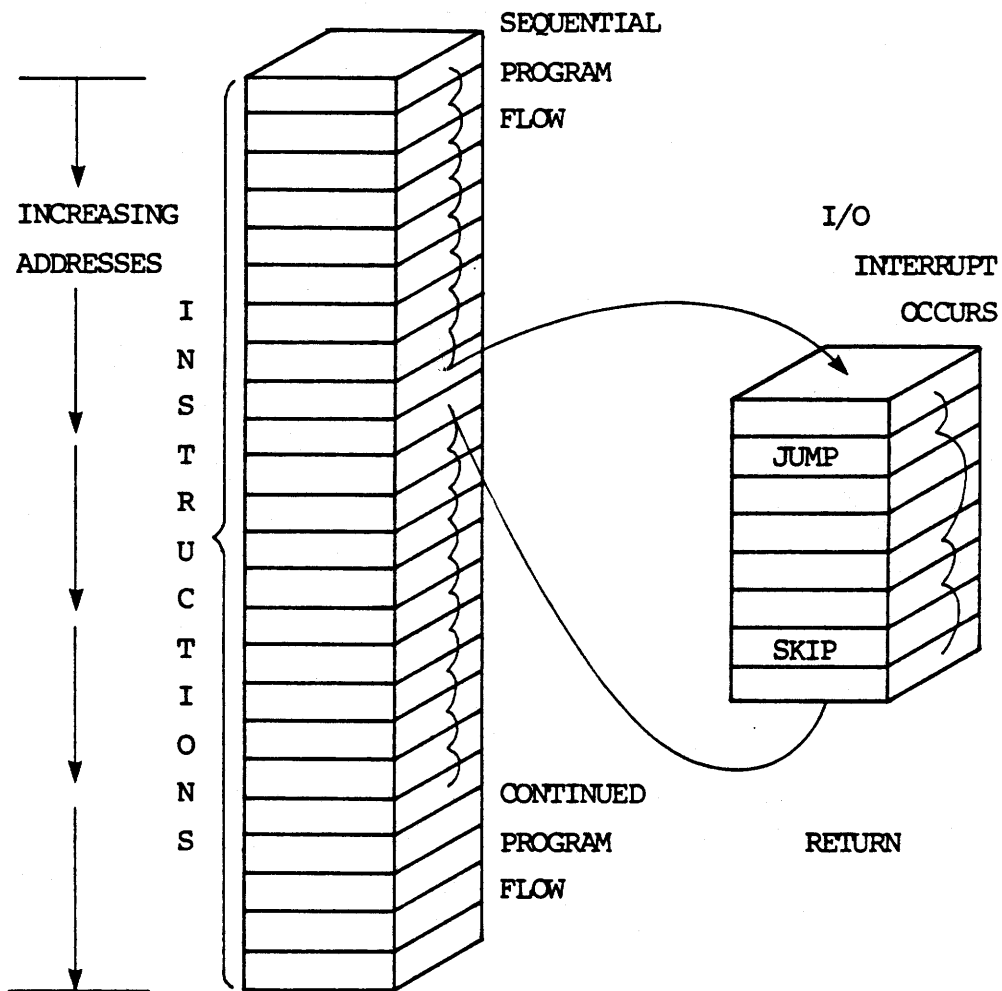
2.2.4 Program Flow Interruption

2.2.4

During the normal running of a program a variety of situations may arise which will make it necessary to interrupt the normal program flow, i.e. to stop ordinary processing temporarily. This may be due to either quite normal occurrences - for instance the necessity of performing an Input/Output operation - or it may be due to exceptional occurrences - external or internal faults or malfunctions.

In both cases the address of the next sequential instruction is saved by the CPU while the interrupt condition lasts. On termination of the interrupt condition the address saved by the CPU is placed in the program counter anew and the interrupted program resumes operation at the correct point in the sequence.

An illustration showing this variation in program flow appears in fig. 2.2.4.



Figur 2.2.4

2.3 Information Formats

2.3

In any computer information is basically represented by some physical quantity - usually electric current or magnetism. The actual nature of this quantity as well as its magnitude carries no importance with respect to use of the computer; the important property is that the relevant quantity can either be present or not present.

2.3.1 Fundamental Concepts

2.3.1

The two possible - but mutually exclusive - states as mentioned above form the basis for all considerations of information processing. The two states are normally indicated by the numerals 0 (zero) and 1 (one) and the nucleus of information thus represented is called a "binary digit" - usually shortened to "bit".

In the RC3603 computer the standard unit of information is however the "word", which is a string of 16 individual bits. As each bit can attain either of two different states, the string of 16 bits can represent $2^{16} = 65,536$ different pieces of information, for instance the integer numbers from 0 up to 65,535. It should here be noted, that although the wellknown mathematical symbolism - i.e. numbers - is often used to describe the information content of a word (or a part of a word), this is in reality only a matter of convenience and does not restrict the actual meaning of the information to this particular subject; nor does it restrict the use to which it may be put. Although the word is the standard unit of information handled by the RC3803 computer it can at times be convenient to subdivide a word into two parts of 8 bits each. Such a half-word is called a "byte" and is capable of representing $2^8 = 256$ different pieces of information.

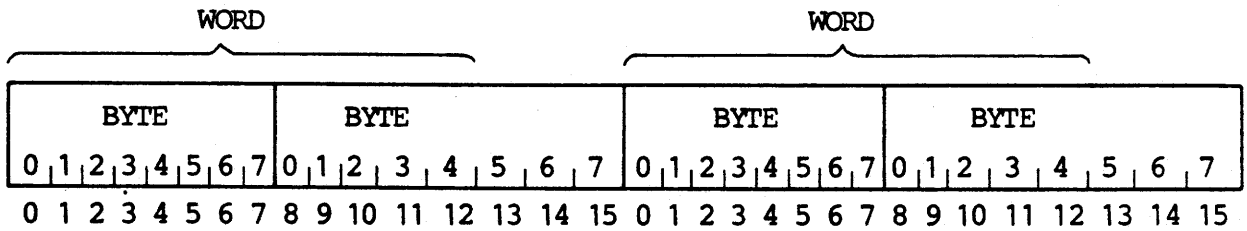
2.3.2 Bit Numbering

2.3.2

When considering the information contained in bytes or words it is convenient to establish a definite method of referencing the individual bits of the byte or word. This is done simply by ordinary numbering of the bits within the word or byte.

The numbering always proceeds from left to right, i.e. the leftmost bit in a word is bit 0 while the rightmost bit in a word is bit 15. Similarly the leftmost bit in a byte is bit 0 while the rightmost bit in a byte is bit 7. Notice that the numbering always starts with bit 0.

The convention adopted here is illustrated in fig. 2.3.2.



Figur 2.3.2

It should also be noted that the adoption of this convention means, that if for instance the word contains a number then the highest-order digit will have the lowest bit number while the lowest-order digit will have the highest bit number.

2.3.3 Binary Representation

2.3.3

If the conventional mathematical notation is adopted by using the numerical values 0 and 1 to indicate the two possible states of the bit, then a word will be read simply as an ordinary 16-digit number - although the number will be written in somewhat unusual manner which in mathematics is called "binary notation".

From our everyday lives we are accustomed to use of numbers in very many contexts; take for instance an arbitrary number like 315. The important feature of a number like this is that the actual value of the individual digit depends on its position in the written number. In effect the way the number is written is just a convenient short-hand way of indicating the magnitude:

$$3 \times 100 + 1 \times 10 + 5 \times 1 = 3 \times 10^2 + 1 \times 10^1 + 5 \times 10^0.$$

This is called "decimal notation" or "base 10" representation because successive digit positions in the number form a sequence of increasing powers of 10.

To indicate that a number is written in base 10 representation a subscript is used whenever there exists a possibility of confusion:

$$315_{10}$$

It is obvious that decimal notation will require ten different symbols to indicate the possible values of the individual digits, namely the symbols: 0, 1, 2, 3, 4, 5, 6, 7, 8, and 9.

Binary notation - or base 2 representation - is in exactly the same way a positional system, the only difference being that in this case successive positions in the number form a sequence of powers of 2. Whereas base 10 representation required ten different symbols for the individual digits base 2 representation will only require two different symbols, namely 0 and 1; this is of course the reason for its dominant position in all aspects of computer technology.

A binary number can of course be used to indicate any magnitude just as well as a decimal number; consequently a binary number can always be converted to the equivalent decimal number and vice versa. Thus:

$$100111011_2 =$$

$$1 \times 2^8 + 0 \times 2^7 + 0 \times 2^6 + 1 \times 2^5 + 1 \times 2^4 + 1 \times 2^3 +$$

$$0 \times 2^2 + 1 \times 2^1 + 1 \times 2^0 =$$

$$1 \times 256_{10} + 0 \times 128_{10} + 0 \times 64_{10} + 1 \times 32_{10} + 1 \times 16_{10} +$$

$$1 \times 8_{10} + 0 \times 4_{10} + 1 \times 2_{10} + 1 \times 1_{10} =$$

$$256_{10} + 32_{10} + 16_{10} + 8_{10} + 2_{10} + 1_{10} =$$

$$315_{10}$$

Internally the CPU will only recognize information given in base 2 representation, but from the example given above it will be clear that the simplicity of binary numbers, owing to the limited number of different symbols used, is counteracted by the necessity of using more digit positions to indicate any given magnitude, i.e. binary numbers tend to become rather long and unwieldy.

Extensive application of binary notation in a manual like this can therefore be somewhat awkward and might even lead to confusion. It cannot be completely avoided, but very often numerical representation to yet another base is used instead.

Noting that a three-digit binary number can represent numerical values from $000_2 = 0_{10}$ to $111_2 = 7_{10}$ it is easily realised, that each group of three bits can be uniquely represented by the eight digits 0, 1, 2, ..., 6 and 7. Therefore the use of a representation to base 8 - so-called octal notation - will retain the basic structure of the binary format, but it will on the other hand only require one third of the positional places needed in pure binary notation.

Expressing the example used on the preceding page in octal notation will yield:

$$315_{10} = 100111011_2 = 473_8.$$

Thus by dividing any string of bits into groups of three and using octal notation a fairly compact and convenient representation is achieved. The subdivision of the string always starts with the rightmost group of three bits and proceeds towards the left. If the number of places in the binary number is not divisible by three the leftmost group will contain only one or two bits. This is however of no particular consequence: conversion to octal notation will take place as outlined above on the additional assumption that the leftmost group is filled-up to three digits by prefixing the necessary one or two zeroes.

2.3.5 Hexadecimal Notation

2.3.5

In some cases still another base is used to represent binary information, namely base 16 - also called hexadecimal notation ("hex"). Just as in the case of octal notation the binary number is formed into groups, but each group will consist of four bits. These four bits can express the numerical values from $0000_2 = 0_{10}$ to $1111_2 = 15_{10}$, and in "hex" it will consequently be necessary to use sixteen individually different symbols for the digits. The numerals from 0 to 9 are of course still used to represent their usual values, whereas the values from 10_{10} to 15_{10} will be represented by the initial six letters of the alphabet: A to F. The example previously used will then yield:

$$315_{10} = 100111011_2 = 473_8 = 13B_{16}.$$

2.4 Numerical Quantities

2.4

The CPU does not intrinsically recognize one type of information as being different from another, but it is quite obvious that in terms of application of the computer numerical quantities do appear in the majority of situations. Numerical quantities basically accepted by the CPU can be either integers or logical quantities.

2.4.1 Integers

2.4.1

Operations on integer quantities can be performed on signed or unsigned binary numbers, which may be carried by the CPU in either single or multiple precision. Single precision integers are two bytes long (16 bits), while multiple precision integers are four or more bytes long.

Unsigned integers use all available bits to represent the magnitude of the number; thus an unsigned, single precision integer can range in value from 0_{10} to $65,535_{10}$ ($2^{16} - 1$) corresponding to the sixteen bits available. Similarly two words taken together as an unsigned, double precision integer can range in value from 0_{10} to $4,294,967,295_{10}$ ($2^{32} - 1$) corresponding to the thirtytwo bits available.

Signed integers use bits 1 to 15 to represent the magnitude of the number while bit 0 is reserved for use as sign bit. The aforesaid assumes single precision; if multiple precision is employed the first (leftmost) word will be structured in this same way while the following word(s) will use all available bits to represent numerical information.

For positive numbers the sign bit is 0 and the remaining bits represent the magnitude of the number in standard binary notation as explained above.

For negative numbers the sign bit is 1 and the remaining bits represent the magnitude of the number in complemented binary notation (also called two's complement form).

Complementing a number - whether in decimal, binary, or any other notation - simply means writing the negative number as the sum of two numbers: a large negative number which is a power of the base plus that positive number which will yield the original number when added to the large negative one. For instance in decimal notation:

$$- 315 = - 1,000,000,000 + 999,999,685.$$

The advantage of this form is that when working within a set number of digit positions, the large negative number will "vanish" - leaving simply a row of zeroes.

To produce the complement - "mechanically" speaking - of a decimal number just subtract the individual digit from 9 to give the digit value of the complement - and then finally add 1 to the last digit.

Thus:

$$\begin{array}{r}
 315_{10} = 0\ 000\ 000\ 100\ 111\ 011 \\
 \quad\quad 1\ 111\ 111\ 011\ 000\ 100 \quad - \text{complementation} \\
 \quad\quad + \underline{\hspace{10em}1} \\
 \\
 - 315_{10} = 1\ 111\ 111\ 011\ 000\ 101
 \end{array}$$

Note that the complementation of a negative number will of course produce the positive of that number.

Complementing zero will produce a carry out of the leftmost bit and leave the number again as zero:

$$\begin{array}{r}
 0\ 000\ 000\ 000\ 000\ 000 \quad - \text{zero} \\
 1\ 111\ 111\ 111\ 111\ 111 \quad - \text{complementation} \\
 + \underline{\hspace{10em}1} \\
 \\
 0\ 000\ 000\ 000\ 000\ 000 \quad - \text{zero}
 \end{array}$$

Note that zero is a positive number!

As shown above complementation of zero will again produce zero and there will thus always be one more negative number than there are non-negative numbers within the given range of digit positions. The numerically largest negative number is a number with the sign bit 1 and all remaining bits 0. The positive value of this number cannot be represented in the same number of digit positions as used to represent the negative number.

Thus a single precision signed integer can lie in the range from - 32,768 to + 32,767 while a double precision signed integer can lie in the range from - 2,147,483,648 to + 2,147,483,647.

Note that addition and subtraction of signed numbers in two's complement form is identical to the same operations on unsigned numbers; the CPU just treats the sign bit as the most significant (highest-order) magnitude bit.

2.4.2 Logical Quantities

2.4.2

Operations on logical quantities can be performed on individual bits, bytes, or words. In all cases the quantities operated on are treated as simple un-structured binary quantities. The logical value "true" is represented by 1 while the logical value "false" is represented by 0. Two logical quantities are identical if and only if they have identical values in corresponding bit positions

The number of bits, bytes, or words operated on will depend on the instruction actually being used.

2.5 Addressing

2.5

It has already been mentioned in the section "Program Execution" (section 2.2.1) that the CPU must be able to locate the instructions stored in main memory. Similarly the CPU must be able to locate the data involved in the operation to be performed - the address of which data will usually be indicated in the instruction.

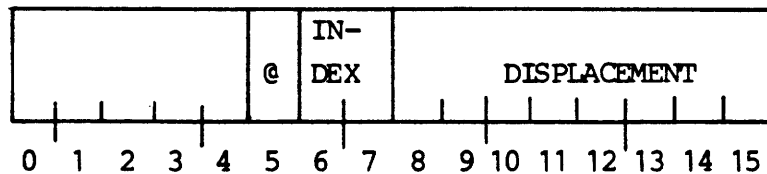
2.5.1 Word Addressing

2.5.1

Main memory is subdivided into a number of words - the actual magnitude of which depends on the CPU configuration actually being employed. Every single word in memory has a definite address, which is given as a number: the first word in memory has the address 0, the next word has the address 1, the next word has the address 2, and so on. It will be recalled that the address of the instruction currently in effect is held in the one-word program counter during the execution of a program. The instruction itself must contain information about the address of data to be used during the execution of that particular instruction.

In contrast to the address held in the program counter the address information contained in the instruction will not always directly specify the necessary address but may form the basis for a calculation whose result will be the desired address. This calculation is called "effective address calculation" and the result of this is the "effective address".

The six instructions which directly reference memory in this way use eleven bits of the word containing the instruction for effective address calculation. The format of these six instructions is shown below:



The eleven bits concerned are bits 5 to 15; of these bit 5 is called the indirect bit, bits 6 and 7 are called the index bits and the remaining eight bits (bits 8 to 15) are called the displacement bits.

There are four essentially different modes of effective address calculation available:

- Page zero addressing
- Relative addressing
- Index Register addressing
- Indirect addressing

2.5.1.1 Page Zero Addressing

2.5.1.1

Page zero addressing is indicated by the index bits being 00. Then the displacement bits are taken as an ordinary unsigned integer number indicating directly the effective address. An 8-bit number will lie in the range from 0 to 255_{10} ; this first block of 256_{10} words in memory, which can be addressed directly in this way, is known as page zero.

2.5.1.2 Relative Addressing

2.5.1.2

Relative addressing is signified by the index bits being 01. In this case the displacement bits are taken as a signed, two's complement integer number. This number is added to the address - contained in the program counter - of the instruction currently in effect; the result of the addition is the effective address. By this means the effective address can be any address in memory accessible to the program as it is defined relative to the address of the instruction. A signed 8-bit number will lie in the range from -128_{10} to $+127_{10}$ relative addressing therefore gives access to a block of 256_{10} words distributed evenly on either side of the instruction.

2.5.1.3 Index Register Addressing

2.5.1.3

Index register addressing is signified by the index bits being either 10 or 11. If they are 10 then accumulator 2 is used as an index register; if they are 11 then accumulator 3 is similarly used.

In both cases the displacement bits are taken as a signed, two's complement integer number; this number is added to the number contained in the accumulator indicated by the choice of index bits. The result of the addition is the effective address.

NOTE: The addition performed in relative and index register addressing is clipped to 15 bits, i.e. the high-order bit (bit 0) of the resulting address is set to 0. For example:
 if the displacement bits are 01 001 111 and (in relative addressing) the program counter stands at 111 111 110 101 011, then the addition should produce the result: 1 000 000 000 011 010, but bit 0 will be set to 0 so that the result reads:
 0 000 000 000 011 010.

If however Memory Extension has been selected the procedure outlined in this note will not apply (for further details see section 5.3).

When index register addressing is used the addition of the displacement to the number contained in the accumulator does not change the value contained in the accumulator.

2.5.1.4 Indirect Addressing

2.5.1.4

While discussing the three addressing modes hitherto covered it has been tacitly assumed, that the indirect bit (bit 5) of the instruction was 0, since only then will the result of the address calculation be the effective address.

If the indirect bit is 1 then the word addressed by either of the three previously mentioned address calculations is expected in itself to contain an address (level 1 indirection). The word concerned will of course contain the usual 16 bits of which now bit 0 will be the indirect bit and bits 1 to 15 will contain the address proper.

If now the indirect bit in the level 1 indirection address is 0 then the address contained in bits 1 to 15 is assumed to be the effective address, but if the indirect bit is 1 then the level 1 indirection address is again expected to contain a further address (level 2 indirection). This procedure will then be repeated until an address is eventually retrieved where bit 0 is 0 and bits 1 to 15 consequently will be taken to be the effective address.

It should be noted that there is no limit to the levels of indirection accepted by the CPU. Neither is there any indication if the chain of indirect addresses due to an error should form a closed loop thus continuing indefinitely.

NOTE: Multi-level indirect addressing mode is disabled if Memory Extension has been selected (section 5.3).

2.5.1.5 Auto Locations

2.5.1.5

Two areas of main memory are reserved for special addressing purposes.

Locations in the range from 20_8 to 27_8 are autoincrement locations, which means that if an indirect addressing chain references an address in this range then the word in that location will be retrieved, the number contained in the word will be incremented by one and this will then be written back into the location. The updated value is then used to continue the chain of indirect addresses.

Locations in the range from 30_8 to 37_8 are autodecrement locations. Exactly the same procedure as outlined above applies here except that the contents of the location will be decremented instead of incremented.

NOTE: When autoincrement or autodecrement locations are referenced in an indirection chain the state of bit 0 before the incrementation or decrementation will be the condition determining the continuation of the chain.
For example:
if an autoincrement location containing the number 177777_8 is referenced during an indirection chain then the next address in the chain will be location 000000_8 - and it will be assumed that this location in itself will contain an address due to the fact, that the original word contained in the autoincrement location (177777_8) had a 1 bit in bit 0.

2.5.2 Byte Addressing

2.5.2

Although the ordinary addressing routines will only allow addressing of complete 16-bit words in memory a convenient programming method is available which will allow handling of individual bytes.

This method involves the use of a "byte pointer" which is a word containing in bits 0 to 14 the address of a normal two-byte word in memory and where bit 15 is the "byte indicator". If the byte indicator is 0 the referenced byte will be the leftmost byte (containing bits 0 to 7) of the word whose address is given in bits 0 to 14 of the byte pointer; if the byte indicator is 1 the referenced byte will correspondingly be the rightmost byte (containing bits 8 to 15).

Programming routines to handle individual bytes in this way are listed in Appendix D of this manual.

Byte addressing cannot be used when locations in the extended memory area are manipulated.

3. Instructions 3.

3.1 Introduction 3.1

The complete set of operation instructions available for RC3803 CPU is divided into four subsets. These are instruction sets for program flow control, data transfer operations, integer arithmetic, and logical operations and a special subset for programming the processor functions plus the optional features: Real Time Clock, Power Fail/Auto-restart, and Memory Extension.

3.2 Instruction Formats 3.2

All instructions in the set are one 16-bit word in length but the lay-out will differ depending on the type of operation to be performed; more specifically this will bear on the number of accumulators employed in the execution of the instruction. In the following description of the different subsets a discussion of the general format in each separate case will appear initially followed by a description of the individual instructions which make up that particular subset.

3.3 Mnemonic Description 3.3

In the description of individual instructions the specific form of the instruction is given in the following generalized format:

MNEMONIC <optional mnemonic> **OPERAND STRING** <optional operands>

The main mnemonic is a group of letter symbols which must be used to initiate the operation concerned in the instruction. To this may in some cases be appended the optional mnemonics, which will cause a modification of the execution of the instruction.

The operand string consists of the actual operands necessary to the execution of the instruction. To this may likewise be appended optional operands.

The symbols <> and == are used as an aid in defining the specific form of each individual instruction:

- < > indicates optional mnemonics or operands
- === used as underlining to identify where definite substitution is required, i.e. where the actual identification of accumulator, address, name, number, or mnemonic must be inserted in the instruction string.

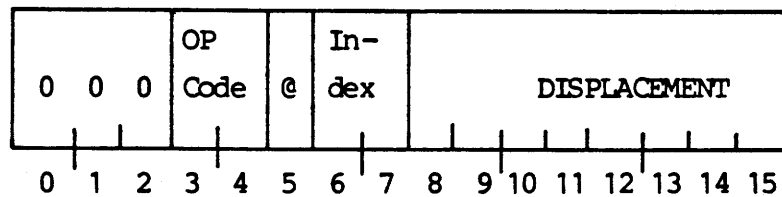
The following abbreviations are used throughout this manual:

- AC Accumulator
- ACD Destination accumulator
- ACS Source accumulator.

3.4 Program Flow Control

3.4

Program flow control operations are handled by way of the program counter - as outlined in section 2.2.1 - and thus do not explicitly utilize any of the available accumulators. The instruction lay-out in this subset is as follows:



In this format bits 0, 1, and 2 are 000, bits 3 and 4 contain the operation code and bits 5 to 15 contain the memory address as described in section 2.5.1.

The symbol @ - placed anywhere in the effective address operand string - will set the indirect bit (bit 5) to 1.

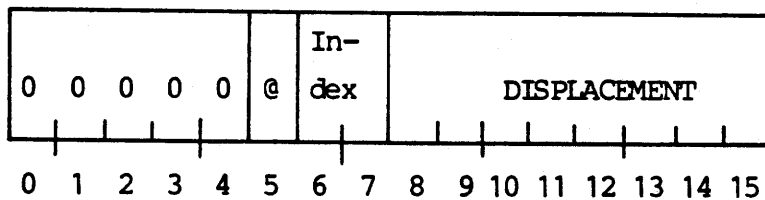
The index bits (bit 6 and 7) are set by a comma followed by one of the digits 0 to 3 as the last operand of the operand string. If no index is coded, the index bits are automatically set to 00. The index bits can be set to 01 by using the character "period" (.) at the beginning of the effective address operand string. When the period is used, it is followed by either a plus or a minus sign and the appropriate displacement, e.g. "+7" or "-2".

The subset contains the following four instructions: JUMP, JUMP TO SUBROUTINE, INCREMENT AND SKIP IF ZERO, and DECREMENT AND SKIP IF ZERO.

3.4.1 JUMP

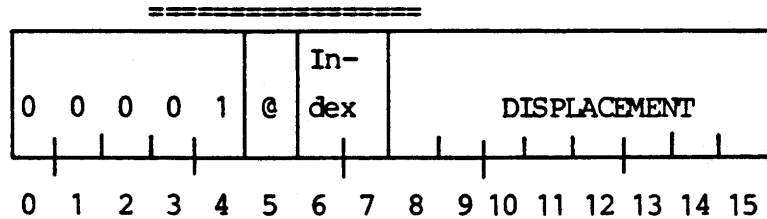
3.4.1

JMP <@> displacement < ,index >



The instruction will cause the effective address to be computed and subsequently placed in the program counter. Sequential operation will then continue with the word addressed by this new value of the program counter.

JSR <@> displacement < ,index >



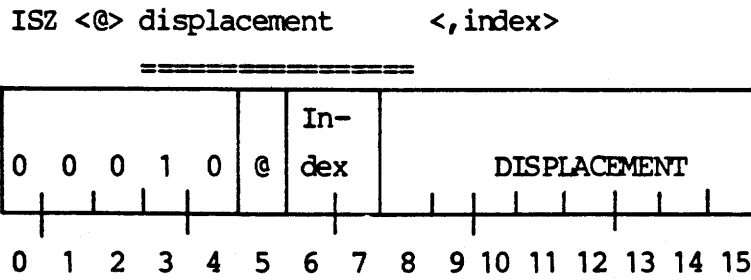
The instruction will cause the effective address to be computed. The current value of the program counter is incremented by one and this number is placed in AC3, whereupon the previously calculated effective address is placed in the program counter and sequential operation then continues with the word addressed by this new value of the program counter.

NOTE: The computation of the effective address is completed before the incremented value in the program counter is written into AC3. This means that if the effective address calculation involves AC3 as an index register, the original value contained in this register will be used in the calculation before it is overwritten with the incremented program counter.

As this instruction saves the incremented value of the program counter in AC3 the use of this instruction for subroutine calls makes the return to the proper point in the main program extremely simple necessitating only the instruction JMP 0,3.

3.4.3 INCREMENT AND SKIP IF ZERO

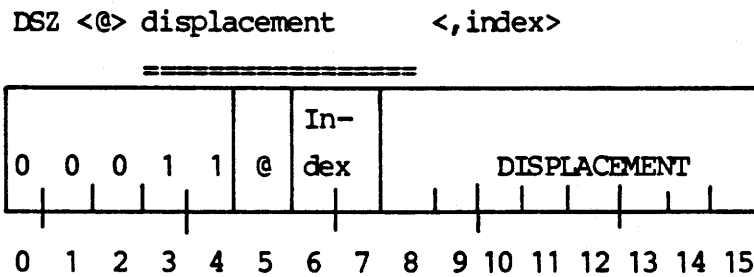
3.4.3



This instruction will cause the effective address to be computed. The word in this location is incremented by one and the result is written back into the original location. If the result of the incrementation is zero then the next sequential instruction is skipped.

3.4.4 DECREMENT AND SKIP IF ZERO

3.4.4



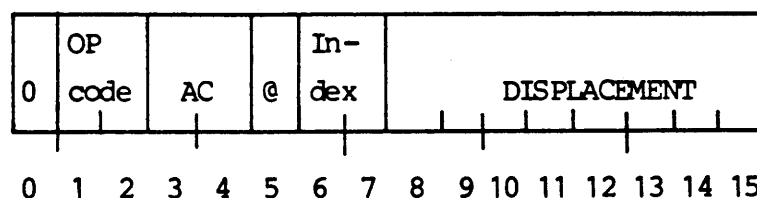
This instruction will cause the effective address to be computed. The word in this location is decremented by one and the result is written back into the location. If the result of the decrementation is zero then the next sequential instruction will be skipped.

3.5 Data Transfer Operation

3.5

Data transfer operations always involve one of the available accumulators as terminal point for the operation (except when the Direct Memory Access feature is utilized, see section 4.5). There are however slight differences in the instruction format depending on whether the data transfer is internal (between main memory and accumulator) or external (between peripheral device and accumulator). This section will only describe the instructions pertaining to internal data transfers, while external transfer will be dealt with in chapter 4: Input/Output.

Internal data transfer instructions use the following lay-out:



In this format bit 0 is 0, bits 1 and 2 contain the operation code, bits 3 and 4 specify the accumulator to be used in the operation, and bits 5 to 15 contain the memory address as outlined in section 2.5.1.

The symbol @ - placed anywhere in the effective address operand string - will set the indirect bit to 1.

The index bits (bits 6 and 7) are set by a comma followed by one of the digits 0 to 3 as the last operand of the operand string. If no index is coded, the index bits are automatically set to 00.

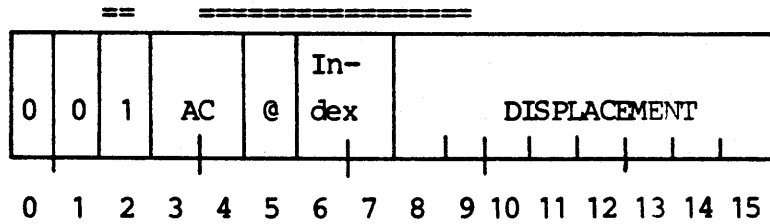
The index bits can be set to 01 by using the character "period" (.) at the beginning of the effective address operand string. When the period is used it is followed by either a plus or a minus sign and the appropriate displacement, e.g. "+7" or "-2".

The internal data transfer subset comprises the following two instructions: LOAD ACCUMULATOR and STORE ACCUMULATOR.

3.5.1 LOAD ACCUMULATOR

3.5.1

LDA ac,<@>displacement <,index>

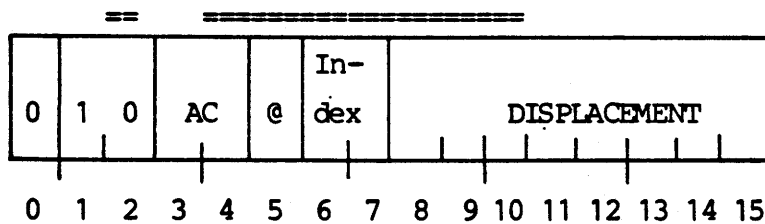


This instruction will cause the effective address to be computed and the word contained in this location will then be retrieved and subsequently written into the accumulator specified. The previous contents of that accumulator will be lost; the contents of the location addressed will remain unchanged.

3.5.2 STORE ACCUMULATOR

3.5.2

STA ac,<@>displacement <,index>

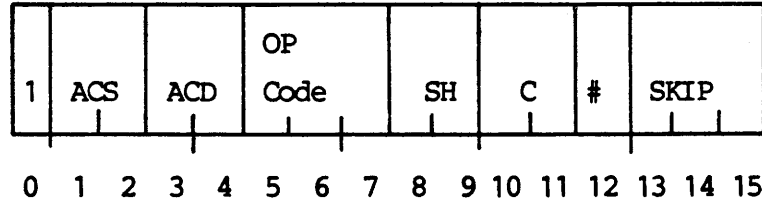


This instruction will cause the effective address to be computed and the word presently located in the accumulator specified will be retrieved and subsequently written into the main memory location indicated by the result of the effective address calculation. The previous contents of this location will be lost; the contents of the accumulator will remain unchanged.

3.6 Integer Arithmetic and Logical Operations

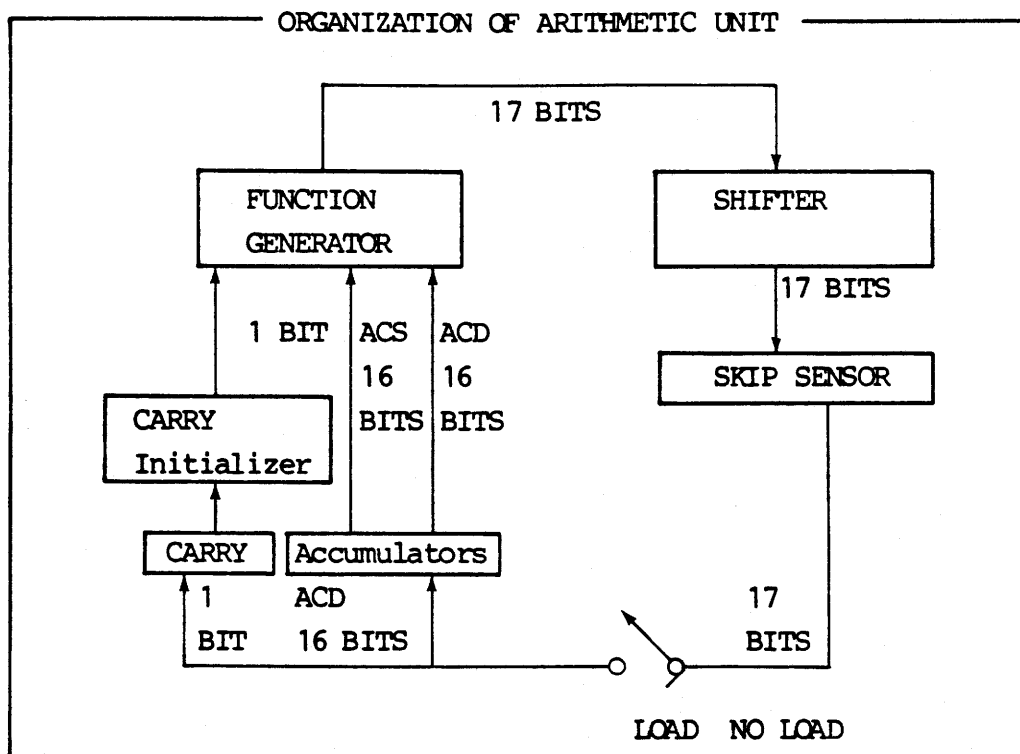
3.6

Arithmetical and logical operations always use two of the available accumulators - usually designated "source accumulator" and "destination accumulator" - to hold the operands involved. Instructions in this subset have the following lay-out:



In this format bit 0 is 1, bits 1 and 2 specify the source accumulator, bits 3 and 4 specify the destination accumulator, bits 5 to 7 contain the operation code, bits 8 and 9 specify the action of the shifter, see fig. 3.6, bits 10 and 11 specify the initializing value of the carry, bit 12 indicates whether the result of the operation must be loaded into the destination accumulator or not, and finally bits 13 to 15 specify the skip test.

All operations initiated by instructions in this subset are performed by way of an arithmetic unit whose logical organisation is illustrated in fig. 3.6:



Figur 3.6

The instruction specifies two accumulators containing the two operands which will have to be supplied to the function generator. This then performs the desired function as specified in bits 5 to 7 of the instruction. In addition to the actual function result the function generator will produce a carry bit, whose value depends on three quantities: an initial value specified by the instruction, the input operands themselves and the function actually performed.

The initial value of the carry bit may be derived from a previous value of same or a completely independent value may be specified via the instruction.

The 17-bit output from the function generator - made up of the carry bit and the 16-bit function result - is then placed in the shifter. Here the 17-bit result can be shifted one place either to the right or to the left; alternatively the two 8-bit halves of the function result can be swapped without affecting the carry bit. The output from the shifter can then be tested for a skip. The skip sensor will test whether the carry bit or the function result itself is equal to zero or not.

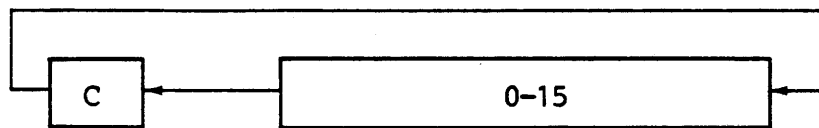
After the skip test the output may be loaded into the carry bit and the destination accumulator respectively. Note however that loading is not an absolute necessity.

The diagrams below illustrate the possible actions taken by the shifter:

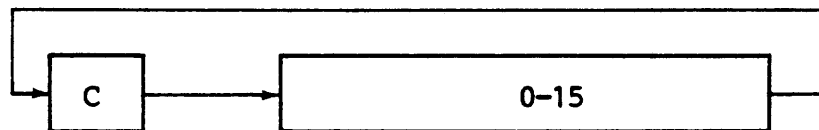
Optional
Mnemonic

Shifter
Operation

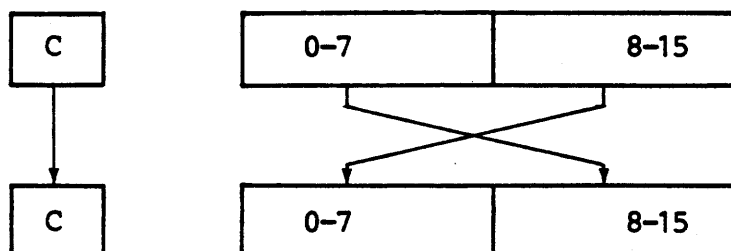
- L All bits are moved one position to the left. Hereby bit 0 is shifted into the carry position while the carry bit is shifted into bit 15.



- R All bits are moved one position to the right. Hereby bit 15 is shifted into the carry position while the carry bit is shifted into bit 0.



- S The two halves of the 16-bit function result change places bit by bit. The carry bit is not affected by this operation.



The following table lists the various options available for use with the instruction format embodying the two-accumulator multiple operation. The characters in the column headed "Class Abbreviation" refer to the specific fields of the instruction format as given at the beginning of this section. The characters in the column headed "Optional Mnemonics" are those which may optionally be appended to the main mnemonic. The binary numbers in the column headed "Bit Settings" show the actual bits which will appear in the appropriate field of the instruction word. The comments in the column headed "Operation" describe the resultant action of the option in question.

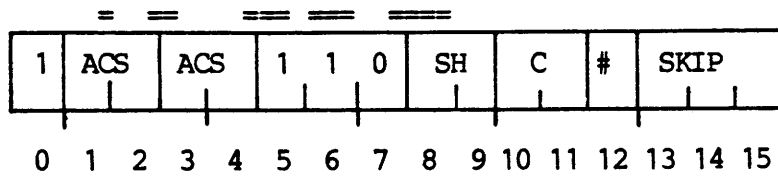
Class Abbreviation	Optional Mnemonic	Bit Settings	Operation
C (Carry Preset)		00	Do not initialize the carry bit.
	Z	01	Initialize the carry bit to 0.
	O	10	Initialize the carry bit to 1.
	C	11	Initialize the carry bit to the complement of its present value.
SH (Shifter)		00	Leave the result of the arithmetic or logical operation unaffected.
	L	01	Combine the carry and the 16-bit result into a 17-bit number and shift it one bit to the left.
	R	10	Combine the carry and the 16-bit result into a 17-bit number and shift it one bit to the right.
	S	11	Exchange the two 8-bit halves of the 16-bit result without affecting the carry bit.
# (Load)		0	Load the result of the shift operation into ACD.
	#	1	Do not load the result of the shift operation into ACD.

SKIP		000	Never skip.
	SKP	001	Always skip.
	SZC	010	Skip if carry equal to zero.
	SNC	011	Skip if carry not equal to zero.
	SZR	100	Skip if result equal to zero.
	SNR	101	Skip if result not equal to zero.
	SEZ	110	Skip if either carry or result equal to zero.
	SBN	111	Skip if both carry and result not equal to zero.

The instruction subset pertaining to integer arithmetic and logical operations include the following instructions: ADD, SUBTRACT, NEGATE, ADD COMPLEMENT, INCREMENT, and MOVE, all of which refer to arithmetical operations, and the logical operations COMPLEMENT and AND.

Integer arithmetic is performed in fixed point mode on 16-bit, signed or unsigned operands in the accumulators. Logical operations are performed on 16-bit unstructured binary operands in the accumulators.

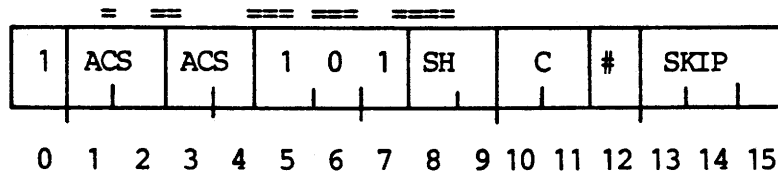
ADD<c><sh><#>acs,acd,<skip>



This instruction will first initialize the carry bit to the specified value. Then the number in ACS is added to the number in ACD and the result is placed in the shifter. If the addition produces a carry = 1 out of the high-order bit (bit 0) the carry bit will be complemented, i.e. this will happen if the sum of the two numbers being added is greater than $65,535_{10}$.

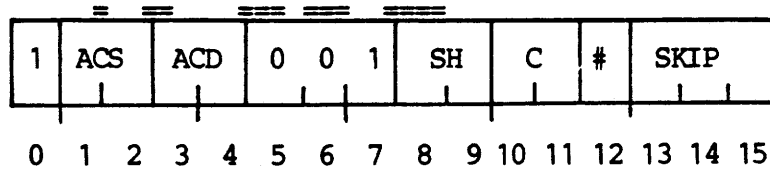
The specified shift operation is then performed and the result of this is placed in ACD provided that the load bit of the instruction has been set to 0. If the skip test demanded results in the condition being true the next sequential instruction will be skipped.

SUB<c><sh><#>acs,acd<,skip>



This instruction will first initialize the carry bit to the specified value. Then the number in ACS is subtracted from the number in ACD (the actual operation being performed by first forming the two's complement of the number in ACS and then adding this to the number in ACD) and the result of the subtraction placed in the shifter. If the operation produces a carry = 1 out of the high-order bit (bit 0) the carry bit will be complemented, i.e. this will happen if the number in ACS is less than or equal to the number in ACD. The specified shift operation is performed and the result of this is placed in ACD provided that the load bit of the instruction has been set to 0. If the skip test demanded results in the condition being true the next sequential instruction will be skipped.

NEG<c><sh><#>acs,acd<,skip,

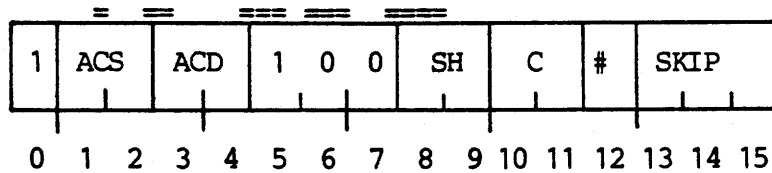


This instruction will first initialize the carry bit to the specified value. Then the two's complement of the number in ACS will be formed and placed in the shifter. If the complementation produces a carry out of the high-order bit (bit 0) the carry bit will be complemented, i.e. this happens if the number in ACS is zero. The specified shift operation is performed and the result of this is placed in ACD provided that the load bit of the instruction has been set to 0. If the skip test demanded results in the condition being true the next sequential instruction will be skipped.

3.6.4 ADD COMPLEMENT

3.6.4

ADC<c><sh><#>acs,acd,<skip>

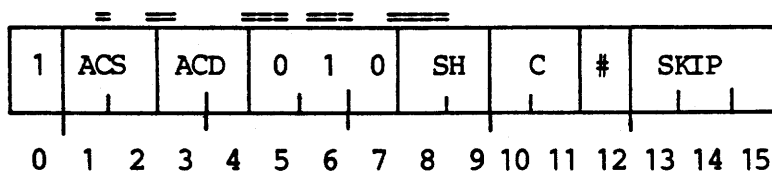


This instruction will first initialize the carry bit to the specified value. Then the logical complement of the number in ACS is added to the number in ACD and the result is placed in the shifter. If the addition produces a carry out of the highorder bit (bit 0) the carry bit will be complemented, i.e. this happens if the number in ACS is less than the number in ACD. The specified shift operation is performed and the result is placed in ACD provided that the load bit of the instruction has been set to 0. If the skip test demanded results in the condition being true the next sequential instruction will be skipped.

3.6.5 MOVE

3.6.5

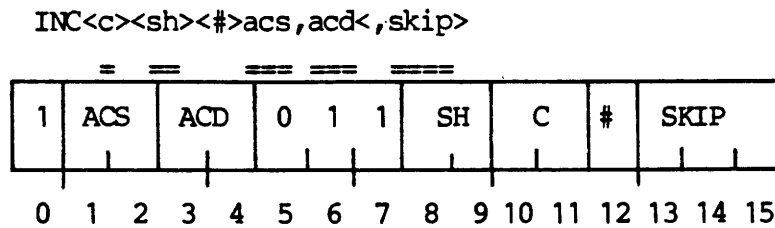
MOV<c><sh><#>acs,acd,<skip>



This instruction will first initialize the carry bit to the specified value. Then the number in ACS is placed in the shifter, the specified shift operation is performed and the result of this is placed in ACD provided that the load bit of the instruction has been set to 0. If the skip test demanded results in the test condition being true the next sequential instruction will be skipped.

3.6.6 INCREMENT

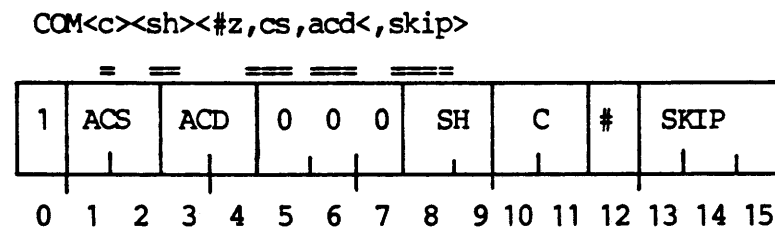
3.6.6



This instruction will first initialize the carry bit to the specified value. Then the number in ACS is incremented by one and the result is placed in the shifter. If the incrementation produces a carry out of the high-order bit (bit 0) the carry bit will be complemented, i.e. this will happen if the number in ACS is 177777_8 . The specified shift operation is performed and the result of this placed in ACD provides that the load bit of the instruction has been set to 0. If the skip test demanded results in the test condition being true the next sequential instruction will be skipped.

3.6.7 COMPLEMENT

3.6.7

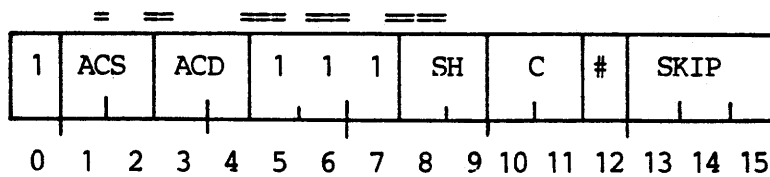


This instruction will first initialize the carry bit to the specified value. The logical complement of the binary quantity in ACS is formed and placed in the shifter. The specified shift operation is performed and the result of this is placed in ACD provided that the load bit of the instruction has been set to 0. If the skip test demanded results in the test condition being true the next sequential instruction will be skipped.

3.6.8 AND

3.6.8

AND<c><sh><#>acs,acd,<skip>



This instruction will first initialize the carry bit to the specified value. Then the logical "and" of the two binary quantities in ACS and ACD is formed and placed in the shifter. Each bit placed in the shifter is 1 if and only if the two corresponding bits in ACS and ACD respectively are both 1; in all other cases the result bit placed in the shifter will be 0. The specified shift operation is performed and the result of this is placed in ACD provided that the load bit of the instruction has been set to 0. If the skip test demanded results in the test condition being true the next sequential instruction will be skipped.

3.6.9 Examples

3.6.9

To show how these different instructions may be used under various circumstances consider the following examples:

3.6.9.1 Deciding the Sign of a Number

3.6.9.1

To determine whether an integer contained in an accumulator is positive or negative can be done in several ways, but the most efficient will be to use the MOVE instruction and thus the inherent power of the two-accumulator multiple-operation format.

Assume that the number in question is contained in AC3. Use of the instruction:

```
MOVL#3,3,SZC
```

will place the number in the shifter and shift the number one place to the left. This will place the original sign bit in the carry bit position and the skip test can then be used to determine whether this bit is 0 or 1. The two following instructions in the program must of course be chosen in such a way that the appropriate action is taken in either case.

Note that by using the optional mnemonic # the load bit is set to 1; thus the output from the shifter will not be loaded back into AC3 and the original number contained herein will therefore be retained for further use.

3.6.9.2 Dividing a Number by a Power of Two

3.6.9.2

To divide a binary number by 2 is simply equivalent to shifting all digits one position to the right (compare with decimal notation where division with 10 - i.e. the base - is readily acknowledged to be produced by this expedient). The fact that the rightmost bit of the original number will be discarded after the shift means that the result of the division will be rounded down to the nearest integer.

The division can be performed simply and efficiently by employing the MOVE instruction as follows:

```
MOVL# 2,2,SZC
MOVOR 2,2,SKP
MOVZR 2,2,SKP
MOVOR 2,2,SKP
MOVOR 2,2
```

The number being divided is supposed to be placed in AC2. The first instruction is simply a repetition of the previous example of deciding the sign of the number. If the number is positive the second instruction will be skipped and operations will continue with the third instruction. This will shift the number one place to the right thus resulting in the division by 2, while at the same time initializing the carry bit to 0, so that when this bit is shifted into the sign bit position the number will remain positive. Note that after division the number is now loaded into AC2 so that this accumulator now holds the result of the division. Finally the fourth instruction is skipped and the fifth repeats the division once more - following which there is no further skip. The repetition means that the end effect will be that the original number has been divided by four. If the number is negative exactly the same sequence of operations are performed with the appropriate alterations to cope with the negative sign - the instructions now in force being the second and fourth.

3.6.9.3 Changing Locations Simultaneously Inverting the Order

3.6.9.

Assume that a block of 30_{10} words, which at present occupy locations 2000_8 to 2035_8 , must be moved to locations 5150_8 to 5205_8 in such a way that the order of the individual words in the block will be inverted.

To do this a section of a program is set up which will autoincrement through one set of locations, autodecrement through the other set and decrement a control count to determine, when the block transfer has been completed. The program section listed below will accomplish this:

```

                LDA    0,CNT    ;comment: set up
                STA    0,21    ;      autoincrement location
                LDA    0,CNT + 1 ;      set up
                STA    0,35    ;      autodecrement location
LOOP:          LDA    0, @ 21  ;      get a word
                STA    0, @ 35 ;      store it
                DSZ    CNT + 2 ;      count down word count
                JMP    LOOP    ;      jump back for next word,
                                skip to here when count
                                is zero
                .
                .
                .
CNT:           001777    ;      1 before source block
+ 1:          005206    ;      1 after destination block
+ 2:           36      ;      word count

```

4. Input/Output

4.

4.1 Introduction

4.1

All useful information processing to be performed by the computer depends on the existence of some means of communication between the CPU and the outside world. For this purpose the CPU is connected to a number of peripheral or Input/Output devices the actual type, size, and number of which is completely independent of the internal logical structure of CPU.

The program must of course contain instructions designed to handle the external data transfer operations; these are all normally termed Input/Output - usually shortened to I/O - operations and allow for the transfer of information in units of bits, bytes, words, or groups of words called "records" depending on the device in use.

All instructions in the I/O subset are basically similar to the previously mentioned internal transfer instructions (section 3.5) except for the fact that addressing as such is not relevant; on the other hand the CPU must have information as to which peripheral unit is to be employed for the actual data transfer and secondly there must be instituted some means of allocating the necessary time for the transfer.

To handle the control of peripheral devices - of which there may be several units of widely differing types connected to the CPU at any given time - the RC3803 CPU is equipped with a six-line device selection network. To initiate operation on a specific device a signal must be transmitted on the selection network, but each individual peripheral device will only respond to this signal if it is identical to the device's own device code. The device code is a six-bit integer number corresponding to the lines in the selection network.

4.2 Operation of I/O Devices

4.2

In general all operations on individual I/O devices are handled by manipulation of two control bits which are called the "Busy" and "Done" flags respectively. If the Busy and Done flags are both 0 the device is idle and cannot perform any operation. To initiate operation on a device the Busy flag must be set to 1, and if the Done flag is not already 0 it must be set to this value. When the device has finished its operation it will itself set the Busy flag to 0 and the Done flag to 1. (If the Busy and Done flags are both - erroneously - set to 1 the situation is meaningless and will produce unpredictable effects.)

Thus to initiate operation on a particular device the program must first determine whether that device is currently performing an operation or not, i.e. it must check the state of the Busy and Done flags. If the Busy and Done flags are 0 and 1 respectively, the program will be able to start the operation by setting Busy to 1 and Done to 0 as described above. When the operation has been completed the device will reset the two flags and will thus be available for another operation whenever necessary.

There are two ways in which the program can test the state of the Busy and Done flags. One is to use the instruction I/O SKIP (cf. section 4.6.7), the other is to employ the Interrupt System which is standard on the RC3803.

4.3 Interrupt System

4.3

The interrupt system consists of an interrupt request line to which each I/O device is connected, an Interrupt On flag in the CPU and a 16-bit interrupt priority mask.

An interrupt is initiated by an I/O device at the time when it completes its operation and resets the Busy and Done flags; simultaneously the device places an interrupt request on the interrupt request line provided that the bit in the interrupt priority mask, which corresponds to the priority level on the device, is 0 (cf. section 4.4). If that particular bit of the mask is 1, the device will still set the flags, but it will not place an interrupt request on the line.

The Interrupt On flag controls the state of the interrupt system in the sense that if the Interrupt On flag is set to 1 the CPU will respond to the process interrupt requests; if the Interrupt On flag is set to 0 it will not do so but will simply go on with normal sequential execution of the program.

The CPU responds to an interrupt request by immediately setting the Interrupt On flag to 0 so that no further interrupts can interfere with the interrupt service routine. The CPU then places the program counter in memory location 0 and executes a "jump indirect" to memory location 1 on the underlying assumption, that this location contains the address - direct or indirect - of the interrupt service routine.

When control has been transferred to the interrupt service routine this routine will first ensure, that the contents of accumulators to be used by the routine are saved, so that these values again can be made available when control is eventually returned to the program proper. The same applies to the carry bit. When this has been accomplished the routine will determine which device requested the interrupt; following this it will proceed with the operations relevant to the servicing of the interrupt.

The determination of which device is in need of service can be accomplished through either the I/O SKIP instruction or the INTERRUPT ACKNOWLEDGE instruction. This last-mentioned instruction returns the six-bit device code of the device requesting the interrupt, thereby initiating operation of that particular device. If more than one device has requested an interrupt, the code returned will be that belonging to the device which is physically closest to the CPU on the I/O bus.

When the I/O device has completed its operation, the interrupt service routine will restore all previously saved values, set the Interrupt On flag to 1 and finally return control to the interrupted program. For this purpose the instruction, that will set the Interrupt On flag to 1, will allow the processor to execute one further instruction before the next interrupt can take place. This extra instruction must be the instruction which returns control to the main program; otherwise the interrupt service routine may go into a loop. However, since the updated value of the program counter - as related above - was placed in location 0 upon responding to the interrupt request, the final instruction in the servicing routine can simply be the instruction "JMP @ 0"; this will transfer control to the main program as intended.

4.4 Priority Interrupts

4.4

If the Interrupt On flag remains 0 throughout the interrupt service routine - as assumed above - all further interrupts will be ignored and there is thus only one level of device priority. This level of priority - i.e. which devices will be able to secure an interrupt - will be determined either by the order in which I/O SKIP instructions are issued or - if the INTERRUPT ACKNOWLEDGE instruction is used - by the relative physical locations on the I/O bus of the various devices.

If the complete computer installation embodies I/O devices of widely differing speeds of operation - such as for example a teletypewriter versus a fixed head disc - it can be convenient for the programmer to set up a multi-level interrupt schedule; this is accomplished by the use of the priority mask coupled with the appropriate instructions.

The priority mask is one 16-bit word to which the individual I/O devices are connected in such a way, that each I/O device is assigned to one specific bit of the mask. The standard mask bit assignment are arranged in such a manner, that devices having roughly the same speed of operation will correspond to the same bit in the mask and will therefore be on the same priority level. (Appendix A of this manual contains - in addition to the device codes - the standard RC mask bit assignments). Although this standard is relevant for most purposes it is not necessary to comply with it, and the programmer is completely free to define his own levels of priority for the individual devices by using the MASK OUT instruction (cf. section 4.7.5). Whenever a bit in the priority mask is set to 1 all devices in the priority level corresponding to that particular bit will be prevented from requesting an interrupt. In addition all pending interrupt requests from devices in that priority level will be ignored.

When multi-level priority handling is implemented, the interrupt service routine must be written in such a way that it may itself be interrupted without damage. This is done by arranging for the main interrupt routine to save the state of the machine, - the contents of the four accumulators, the carry bit, and the return address - whenever it takes over control.

The information concerned must be stored in separate locations for each time the interrupt handler is entered, so that a higher level of interrupt will not overlay the return information corresponding to a lower priority level. Having thus saved the necessary return information the main interrupt routine must determine which device has requested service and then transfer control to the correct interrupt handling routine. The actual transfer is effected in the same way as for the previously described single-level interrupt handler.

When the correct service routine has received control it will save the current priority mask, establish the new priority mask and activate the interrupt system. When it has finished servicing the I/O device, the routine will de-activate the interrupt system, reset the priority mask to its original form, restore the state of the machine, again activate the interrupt system, and finally return control to the interrupted program.

4.5 Direct Memory Access Data Channel

4.5

The handling of data transfers under program control as described above requires an interrupt plus the execution of several instructions for each word transferred and therefore occupies valuable time on the processor.

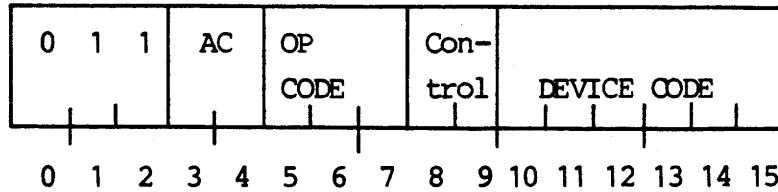
To avoid this and at the same time to obtain higher transfer rates the RC3803 CPU is equipped with a separate data channel through which an I/O device - at its own request - can gain direct access to main memory.

When an I/O device is ready to send or to receive data it requests access to memory via the data channel. All such requests are synchronized by the CPU at the beginning of each memory cycle. The CPU will then pause at specified points during the execution of an instruction; at each pause it will accept all previously synchronized requests in which instance a word will be transferred directly via the channel from the device to memory or vice versa without interference with the program.

All requests are honoured in relation to the relative physical positions on the I/O bus of the different requesting devices; that is: the device being physically closest to the CPU is serviced first, then the next closest device and so on until all requests have been processed. As synchronization of new requests occur continuously even while previous requests are being attended to, a device can in effect saturate the channel if it requests transfer continually. All devices further out on the bus cannot gain access to the channel until the transfers involving the closer device have been processed, although of course devices which are closer still on the bus will not be affected.

In addition to the pause intervals during the execution of an instruction data channel request will be handled on completion of an instruction. At this point furthermore, all outstanding I/O interrupt requests will be accepted. When all such data transfers have been accomplished the CPU will continue with normal sequential operation.

All I/O instructions use the format given below:



In this format bits 0, 1, and 2 are 011, bits 3 and 4 specify the accumulator involved, bits 5 to 7 contain the operation code, bits 8 and 9 control the Busy and Done flags in the device, and bits 10 to 15 contain the device code. The six bits provided for the device code will define 64_{10} unique devices, but the total number of separate devices which can be employed simultaneously on any given installation will be slightly lower than this as some of the available device codes are reserved for the CPU and certain processor features. Of the remaining codes some have been assigned to specific devices by Regnecentralen. A complete listing of device codes appear in Appendix A.

The subset of I/O instructions has a number of options that can be obtained by appending the appropriate optional mnemonic to the standard mnemonic of the instruction. These optional mnemonics are listed in the table below; the column headings correspond to those given in section 3.6.

Class Abbreviation	Optional Mnemonic	Bit Settings	Operation
F (Flags)		00	Does not affect the Busy and Done flags.
	S	01	Start the device by setting Busy = 1 and Done = 0.
	C	10	Idle the device by setting both Busy and Done to 0.
	P	11	Pulse the special in-out bus control line. The effect - if any - depends on the actual device.

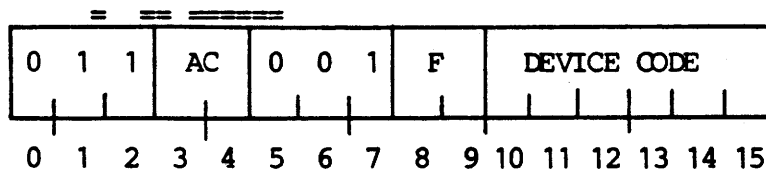
T (Tests)	BN	00	Tests for Busy = 1.
	BZ	01	Tests for Busy = 0.
	DN	10	Tests for Done = 1.
	DZ	11	Tests for Done = 0.

The I/O instruction subset contains the following instructions: DATA IN A, DATA IN B, DATA IN C, DATA OUT A, DATA OUT B, DATA OUT C, I/O SKIP, and NO I/O TRANSFER.

4.6.1 DATA IN A

4.6.1

DIA<f> ac,device



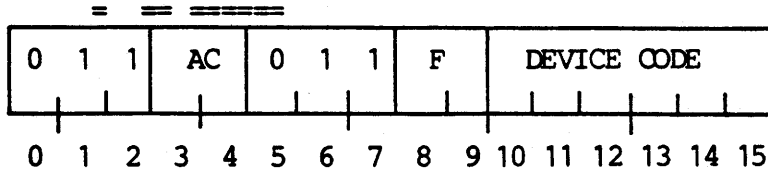
This instruction will place the contents of the A input buffer on the specified device in the AC specified in the instruction. After the data transfer has been completed the Busy and Done flags are set as specified by "f".

The number of data bits moved depends on the size of the buffer and the mode of operation of the device selected. Bits in the AC not receiving any data are set to 0.

4.6.2 DATA IN B

4.6.2

DIB<f> ac,device

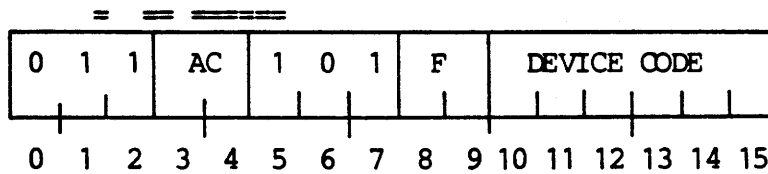


This instruction will have exactly the same effect as the one previously described - except that it will utilize the B buffer of the peripheral device.

4.6.3 DATA IN C

4.6.3

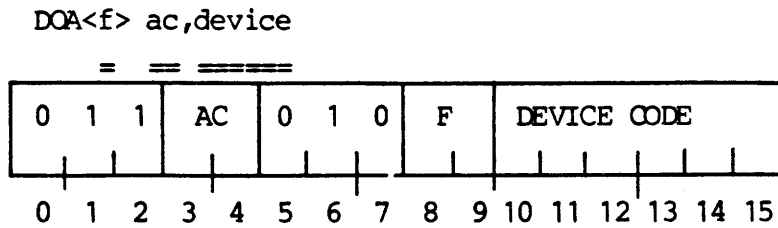
DIC<f> ac,device



This instruction will have exactly the same effect as the two previously described - except that it will utilize the C buffer of the peripheral device.

4.6.4 DATA OUT A

4.6.4

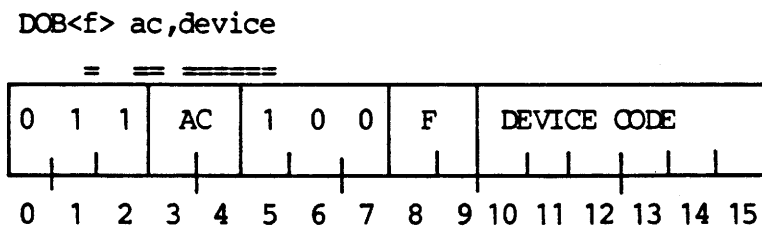


This instruction will place the contents of the specified AC in the A output buffer of the selected device. After the data transfer has been completed, the Busy and Done flags are set as specified by "f". The contents of the AC will remain unaltered.

The number of data bits moved will depend on the size of the buffer and on the mode of operation of the device.

4.6.5 DATA OUT B

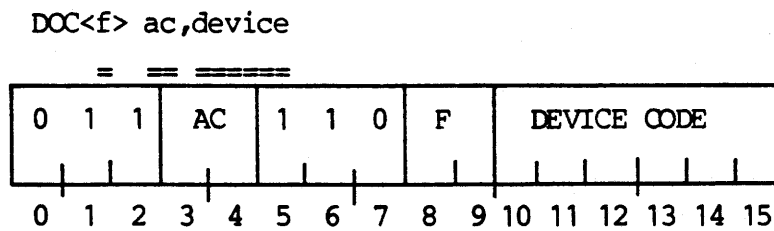
4.6.5



This instruction will have exactly the same effect as the one previously described - except that it will utilize the B buffer of the peripheral device.

4.6.6 DATA OUT C

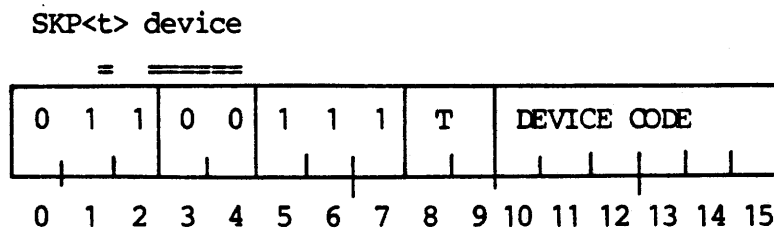
4.6.6



This instruction will have exactly the same effect as the two previously described - except that it will utilize the C buffer of the peripheral device.

4.6.7 I/O SKIP

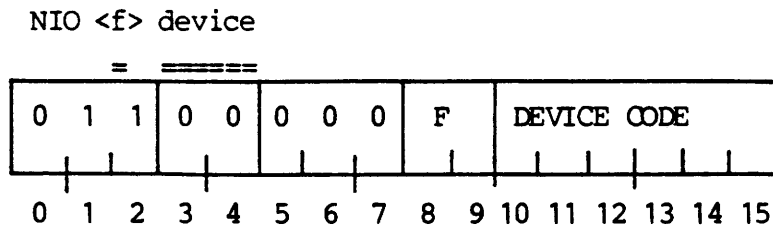
4.6.7



This instruction will test the state of the Busy and Done flags and will thus enable the programmer to decide on actions to be taken in consequence of the values of these flags, i.e. whether a device is in need of service from the interrupt system or not. The test performed depends on the value of bits 8 and 9 of the instruction and is selected by appending the appropriate optional mnemonic to the instruction according to the table given in section 4.6. If the test condition specified by "T" is true the next sequential instruction will be skipped.

4.6.8 NO I/O TRANSFER

4.6.8



This instruction will set the Busy and Done flags in the selected device according to the control code specified by "F".

4.7 Central Processor Functions

4.7

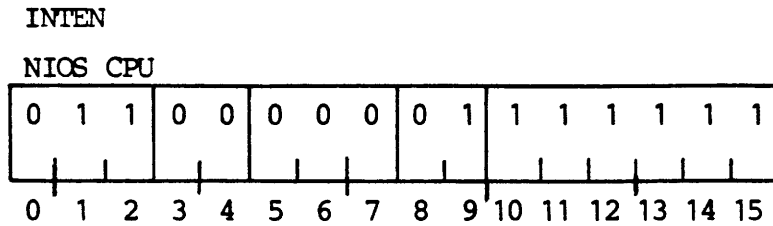
I/O instructions with a device code of 77_8 will perform a number of special functions rather than control a specific peripheral device. With the exception of the I/O SKIP instruction all I/O instructions having a device code of 77_8 will use bits 8 and 9 of the instruction format to control the state of the Interrupt On flag. The I/O SKIP instruction - when used with a device code of 77_8 - will cause a test of the state of the Interrupt On flag. (Alternatively it may be used to test the state of the Power Fail flag; see section 5.2). The optional mnemonics for these special instructions are the same as for normal I/O instructions. The table below lists the resulting actions for these instructions when used with the special device code 77_8 .

Class Abbreviation	Optional Mnemonic	Bit Settings	Operation
F (Flags)		00	Does not affect the state of the Interrupt On flag.
	S	01	Set the Interrupt On flag to 1.
	C	10	Set the Interrupt On flag to 0.
	P	11	Does not affect the state of the Interrupt On flag.
T (Tests)	BN	00	Tests for Interrupt On = 1.
	BZ	01	Tests for Interrupt On = 0.
	DN	10	Tests for Power Fail = 1.
	DZ	11	Tests for Power Fail = 0.

In addition to use of the ordinary I/O instructions with the special device code 77_8 , there is a subset of special instructions for processor functions which contains the following instructions: INTERRUPT ENABLE, INTERRUPT DISABLE, READ SWITCHES, INTERRUPT ACKNOWLEDGE, MASK OUT, I/O RESET, HALT, and CPU SKIP.

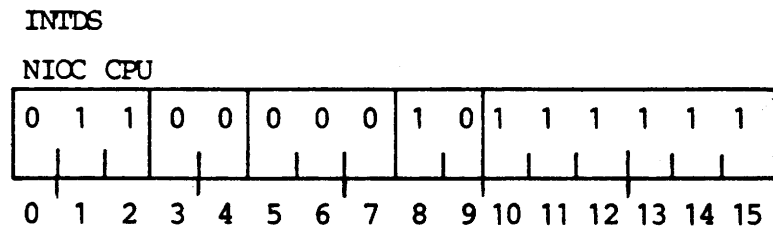
4.7.1 INTERRUPT ENABLE

4.7.1



This set of instructions will set the Interrupt On flag to 1. If the state of the Interrupt On flag is hereby changed, the CPU will allow one more instruction to be executed before the first I/O interrupt can occur.

4.7.2

4.7.2 INTERRUPT DISABLE

This set of instructions will set the Interrupt On flag to 0.

4.7.3 READ SWITCHES

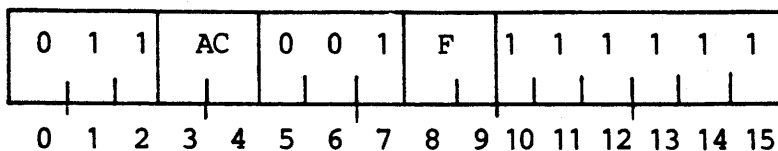
4.7.3

READS ac (F = 00)

==

DIA <f> ac,CPU

= =



This set of instructions will place the current setting of the data switches on either the Diagnostic Front Panel (if connected) or the front frame of the CPU-board in the AC specified in the instructions. After the transfer has been completed, the Interrupt On flag is set according to the control code specified by "F".

4.7.4 INTERRUPT ACKNOWLEDGE

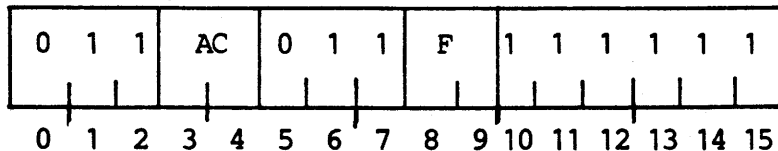
4.7.4

INIA ac (F = 00)

==

DIB <f> ac,CPU

= =



This set of instructions will cause the six-bit device code of that device, which is physically closest to the CPU on the I/O bus, to be placed in bits 10 to 15 of the AC specified in the instructions. Bits 0 to 9 of the AC involved will be set to 0. After the transfer has been completed the Interrupt On flag is set according to the control code specified by "F".

4.7.5 MASK OUT

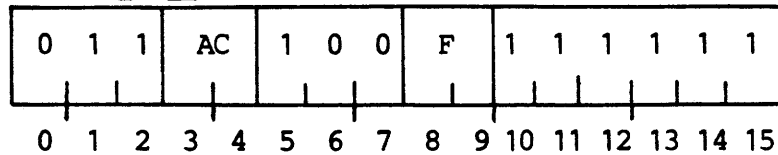
4.7.5

MSKO ac (F = 00)

==

DOB <f> ac,CPU

= =



This set of instructions will place the contents of the AC specified in the priority mask. After the transfer has been completed, the Interrupt On flag is set according to the control code specified by "F". The contents of the AC remain unaltered.

NOTE:

The digit 1 in any bit position disables interrupt requests from any peripheral device in the corresponding priority level.

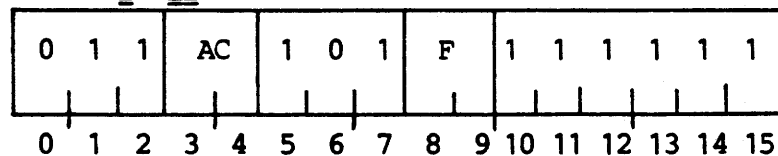
4.7.6 I/O RESET

4.7.6

IORST (F = 10)

DIC <f> ac,CPU

= =



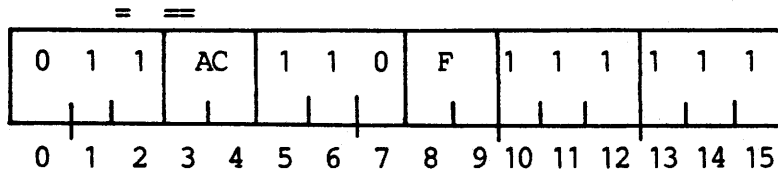
This set of instructions will cause the Busy and Done flags in all I/O devices to be set to 0; simultaneously all bits in the 16-bit priority mask are set to 0. The Interrupt On flag is set according to the control code specified by "F".

4.7.7 HALT

4.7.7

HALT (F = 00)

DOC <f> ac,CPU

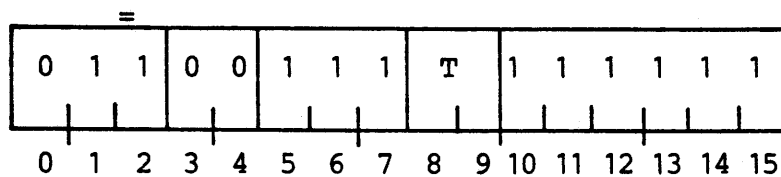


This set of instructions will set the Interrupt On flag according to the control code specified by "F". Following this the processor is stopped.

4.7.8 CPU SKIP

4.7.8

SKP <t>, CPU



This instruction will cause the Interrupt On flag or the Power Fail flag to be tested depending on the control code specified by "T". If the test condition is true the next sequential instruction will be skipped.

5. Processor Features

5.

5.1 Introduction

5.1

Features included in the RC3803 computer are a power monitor which will handle automatic shut-down and restart in the event of a failure of the power supply, a special CPU function allowing memory to be extended beyond the 32K words' capacity, and an extended instruction set containing the time consumption routines in the RC3600 software.

The extended instruction set covers a set of micro-programmed monitor-procedures.

Each procedure is described in details using a pseudo-language notation as explained below.

A micro-programmed monitor-procedure is in fact one instruction possibly interrupted by interrupt or DMA-request.

When finished request the procedure is restarted, that is the instruction is executed once again. If then terminated the next instruction is executed as usual. This formalisme is described using the pseudo-functions `fetchnext` and `serverrequest`:

Fetchnext: The actual value of the instruction counter PC is incremented and the next instruction is fetched from the memory word addressed in PC.

Serverrequest: The actual value of the instruction counter PC is decremented and the request sevice is entered. When finished the `serviceroutine` includes a call of `fetchnext`, hereby initiating execution of the actual instruction once again. Please refer to section 4.3 for more details.

The notation used in describing the implemented procedures and the examples given are related to those given in the system manual 'MUS SYSTEM, Programming Guide, Rev. 1.00', with the following notice:

CUR = Current process description address.

PC = Program Counter (instruction counter).

Core memory in the RC3803 computer is of magnetic type and information stored in it is therefore independent of power supply and will be retained unaltered for a very considerable time in event of the power supply being cut off. The same does not, however, apply to the accumulators, program counter, various flags, etc. in the CPU; all values in these components will be indeterminate following a break in the supply of power. The Power Fail feature provides the capability to overcome this difficulty. In the event of an unexpected power failure the voltage will rapidly decrease from its normal value to the value where the processor automatically shuts down completely. There will however be an interval of time - roughly one or two milliseconds - between the initial drop-off of voltage and the actual shut-down. The Power Fail circuit will sense the beginning reduction of voltage, set the Power Fail flag and request an interrupt. The interrupt service routine will then be able to utilize the interval before shut-down to store the contents of the accumulators, the carry bit, and the current priority mask in memory. In addition to this it will save memory location 0, where it will store a jump instruction to the desired restart location and finally it will execute a HALT. As one or two milliseconds is sufficient time to execute up to 1500 instructions there is ample time to perform the power fail routine.

When the power supply is again restored, the CPU will execute a "JMP 0" instruction after an interval of 100 milliseconds. This will effect a restart of the interrupted program.

The power fail feature has no device code and no interrupt disable bit in the priority mask. Neither does it respond to the INTERRUPT ACKNOWLEDGE instruction. The Power Fail flag can be tested by means of the CPU SKIP instruction as described in section 4.7.8.

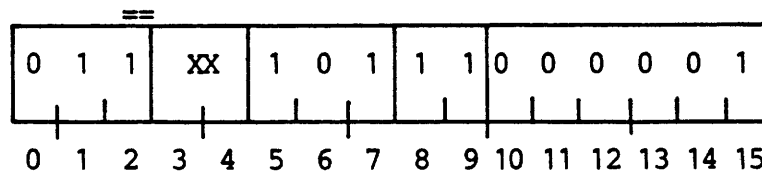
5.3 Memory Extension

5.3

Normal memory capacity of the RC3803 computer is 32K words (64K bytes). The Memory Extension feature provides the capability to increase this capacity to 64K words (128K bytes).

To switch from running in normal configuration to running in extended memory configuration the following instruction must be applied:

DICP ac, 1

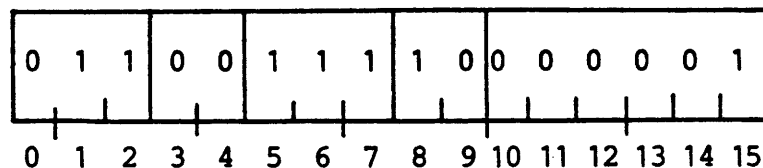


X = DON'T CARE

This instruction will allow the CPU to utilize the extra block of core memory and it will furthermore set the Memory Extension flag to 1. For the instruction to have the desired effect the switch 64K/128K BYTES on the front frame of the CPU-board must be in the 128K BYTES position; otherwise the instruction is dummy.

The state of the Memory Extension flag can be tested with the I/O SKIP instruction using the device code (001) reserved for the Extended Memory (see Appendix A). The testing of the flag thus follows through the instruction:

SKPDN 1



As usual with this instruction the next sequential instruction will be skipped if the test condition is true, i.e. if the Memory Extension flag is 1.

If the 64K/128K BYTES switch on the front panel is returned to the 64K BYTES position the Memory Extension flag is not automatically set back to 0 (although the CPU no longer will be able to utilize the extended memory block). To return the Memory Extension flag to 0 an I/O RESET instruction must be used. The flag will also be set to 0 following a power up.

The CPU can execute programs placed in all 128K bytes, because Multi-level indirect addressing is disabled, when Memory Extension is selected.

The Disc Controller is capable of writing data into and reading data from the extended area of memory.

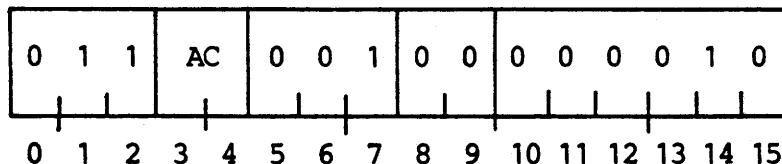
NOTE: It is important to be aware of the fact, that when Memory Extension is applied the program counter will continue from 77777_8 to 100000_8 in the course of normal sequential operation.

5.4 CPU Identify

5.4

IDFY ac

==



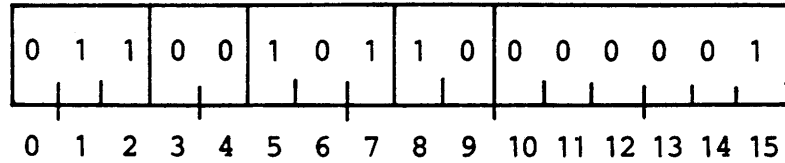
This instruction loads a microprogram revision number (2 for RC3803) into the accumulator selected in the AC-field.

5.5 Byte Manipulation

5.5

In addition to performing operations on structured and unstructured 16-bit quantities, the instruction set of the RC3803 allows loading and storing of 8-bit bytes.

LDB



CALL:	RETURN:
; AC0 -	AC0(0:7):=0; AC0(8:15):= BYTE
; AC1 FROM BYTEADDRESS	UNCHANGED
; AC2 -	UNCHANGED
; AC3 -	UNCHANGED

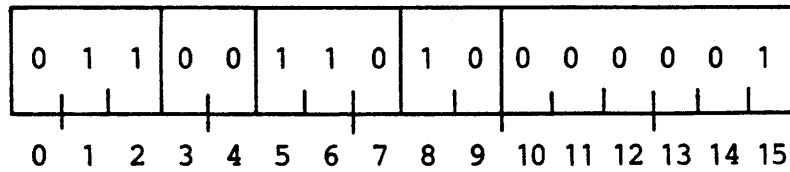
The 8-bit byte addressed by the byte pointer contained in AC1 is placed in bits 8-15 of the AC0. Bits 0-7 of the AC0 are set to 0. The contents of AC2 and AC3 remain unchanged.

The byte address in AC1 is a word address left shifted one and with a one added in bit 15 if the byte addressed within the word is placed in bit 8:15.

5.5.2 STORE BYTE

5.5.2

STB



	CALL:	RETURN:
; AC0	AC0(8:15):= BYTE	UNCHANGED
; AC1	TO BYTEADDRESS	UNCHANGED
; AC2	-	UNCHANGED
; AC3	-	UNCHANGED

Bits 8-15 of AC0 are placed in the byte addressed by the pointer contained in AC1. Bits 0-7 of AC0 are don't care and not affected.

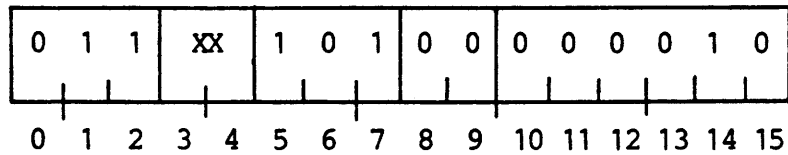
The contents of AC2 and AC3 remain unchanged. Note that the remaining part of the word addressed is untouched.

The byte address in AC1 is a word address left shifted one and with a one added in bit 15 if the byte addressed with the word placed in bit 8:15.

5.6 Byte Move

5.6

BMOVE



X = DON'T CARE

	CALL:	RETURN:
; AC0	CONVERT ADDR.	CONVERT ADDR
; AC1	FROM ADDR.	FROM ADDR + BYTE COUNT + 1
; AC2	TO ADDR.	TO ADDR + BYTE COUNT + 1
; AC3	BYTE COUNT	ZERO

This instruction moves a byte string from the byte address specified in AC1 to the byte address specified in AC2. The number of bytes to be moved is specified in AC3. If AC0 \neq 0 the moved byte is converted via a table addressed by AC0.

The byte addresses in AC1, AC2, and AC0, are word addresses shifted one and with a one added in bit 15 if a right byte is addressed.

The instruction may be interrupted by interrupt request and data channel request following the algorithm:

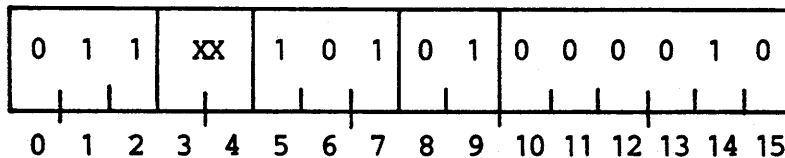
```

START,                                     ; BMOVE :
LOOP:   If bytecount = 0 then goto EXIT else
        begin
            Q:= byte (fromaddr)
            If convertaddr <> 0
                then Q:= byte (Q + convertaddr) ;
            byte (toaddr):= Q
            fromaddr := fromaddr + 1           ; Update AC1
            toaddr   := toaddr + 1           ; Update AC2
            bytecount := bytecount -1       ; Update AC3
        end ;
TEST:   If (INT REQ or DMA REQ) = 0
        then goto LOOP
WAIT:   Servereq (PC)                       ; Dcr. prog. counter and
                                             serve req.
EXIT:   Fetchnext (PC)                     ; Incr. prog. counter and
                                             exec. instr.

```

5.7 Word Move 5.7

WMOVE



X = DON'T CARE

	CALL:	RETURN:
; AC0	WORD COUNT	ZERO
; AC1	FROM ADDR.	FROM ADD + WORD COUNT + 1
; AC2	TO ADDR.	TO ADDR + WORD COUNT + 1
; AC3	-	UNCHANGED

This instruction moves a word string from the address in AC1 to the address in AC2. The number of words to be moved is specified in AC0.

The instruction may be interrupted by interrupt and data channel request following the algorithm:

```

START,                                     ; WMOVE :
LOOP:   If wordcount = 0 then goto EXIT else
        begin
            Q:= word (fromaddr)
            word (toaddr):= Q
            fromaddr := fromaddr + 1      ; Update AC1
            toaddr   := toaddr + 1       ; Update AC2
            wordcount := wordcount -1    ; Update AC0
        end
TEST:   If (INT REQ or DMA REQ) = 0
        then goto LOOP
WAIT:   Servereq (PC)                    ; Decr. PC and serve req.
EXIT:   Fetchnext (PC)                   ; Incr. PC and exec instr.

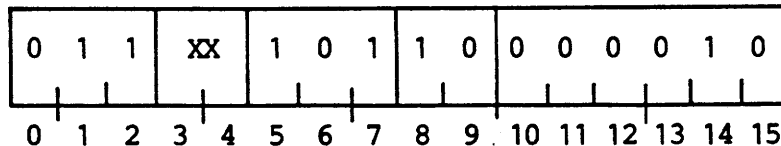
```

5.8

Search Item

5.8

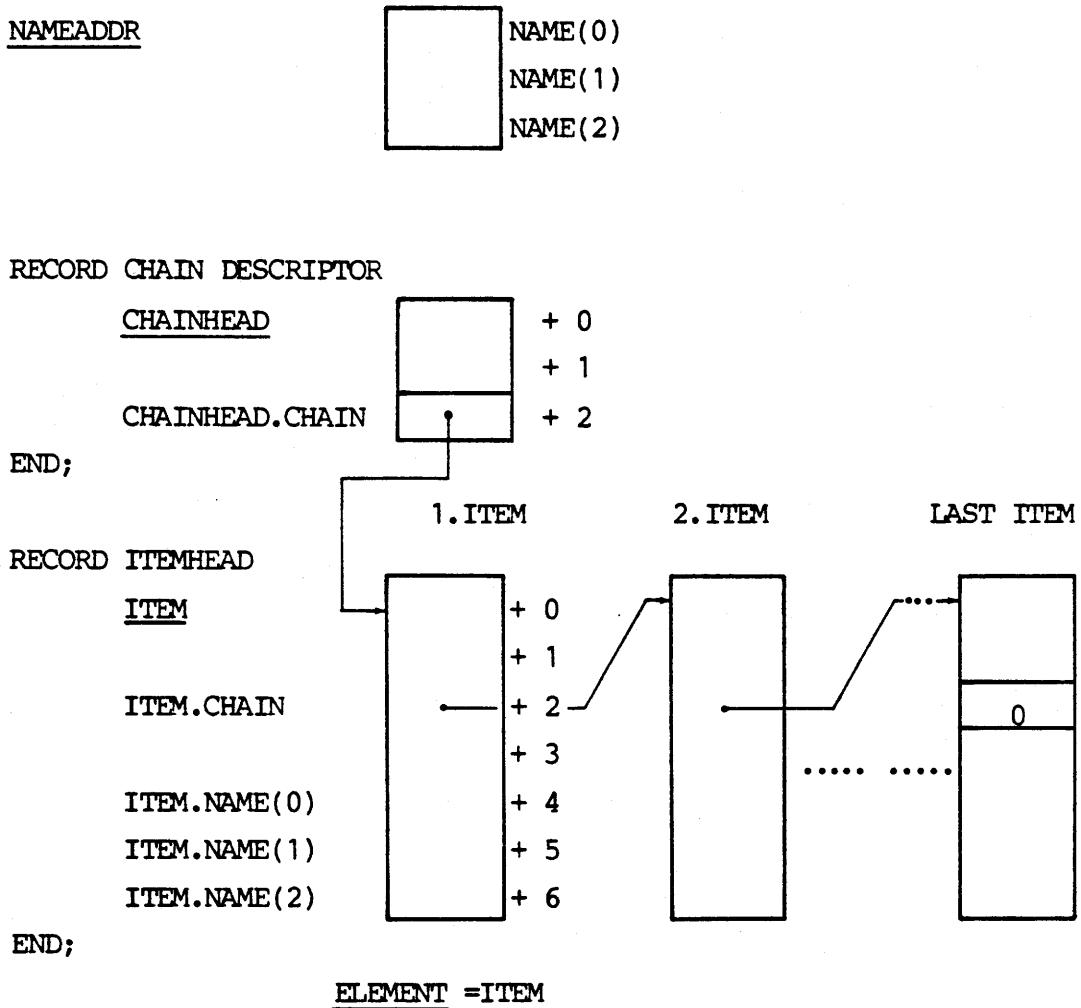
SCHEL



X = DON'T CARE

	CALL:	RETURN:
; AC0	-	DESTROYED
; AC1	CHAINHEAD	DESTROYED
; AC2	NAMEADDR	ELEMENT
; AC3	-	CUR

This instruction searches the chain for an element with a given name and delivers the address of the element, if found, and a zero if the name is not found in the chain. The chain-datastructure is illustrated in fig. 5.8.



EXAMPLE: Search by name through the DOMUS - coreitemchain.

Figur 5.8

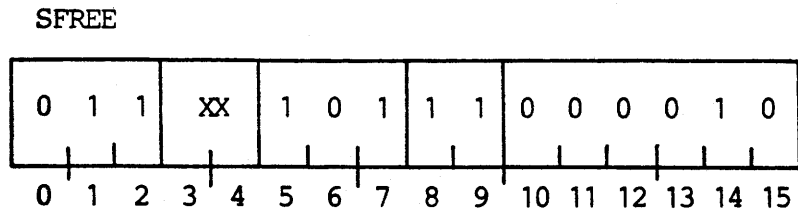
The instruction may be interrupted by interrupt and data channel request following the algorithm:

```

START:  element:= CHAINHEAD           ; SCHEL:
LOOP:   element:= word (element.chain) ; AC3:= next element
        If element = 0 then goto EXIT
        Q:= word (nameaddr)
        Q1:= word (element.name)      ; Compare 1. word
        If Q <> Q1 then goto TEST     ; in name
        Q:= word (nameaddr + 1)
        Q1:= word (element.name + 1)  ; Compare 2. word
        If Q <> Q1 then goto TEST     ; in name
        Q:= word (nameaddr + 2)
        Q1:= word (element.name + 2)  ; Compare 3. word
        If Q <> Q1 then goto TEST     ; in name
        Goto EXIT
TEST:   CHAINHEAD:= element           ; Saved work value for use
        If (INT REQ or DMA REQ) = 0   ; when restarted after int.
            then goto LOOP
WAIT    Servereq (PC)                 ; Dcr. PC and servereq
        Fetchnext (PC)                ; Incr. PC and exec instr.
        ; *
EXIT:   AC2:= element                 ; AC2:= AC3
        AC3:= CUR                     ; AC3:= CUR,
        Fetchnext (PC)                ; Incr. PC and exec. instr.

```

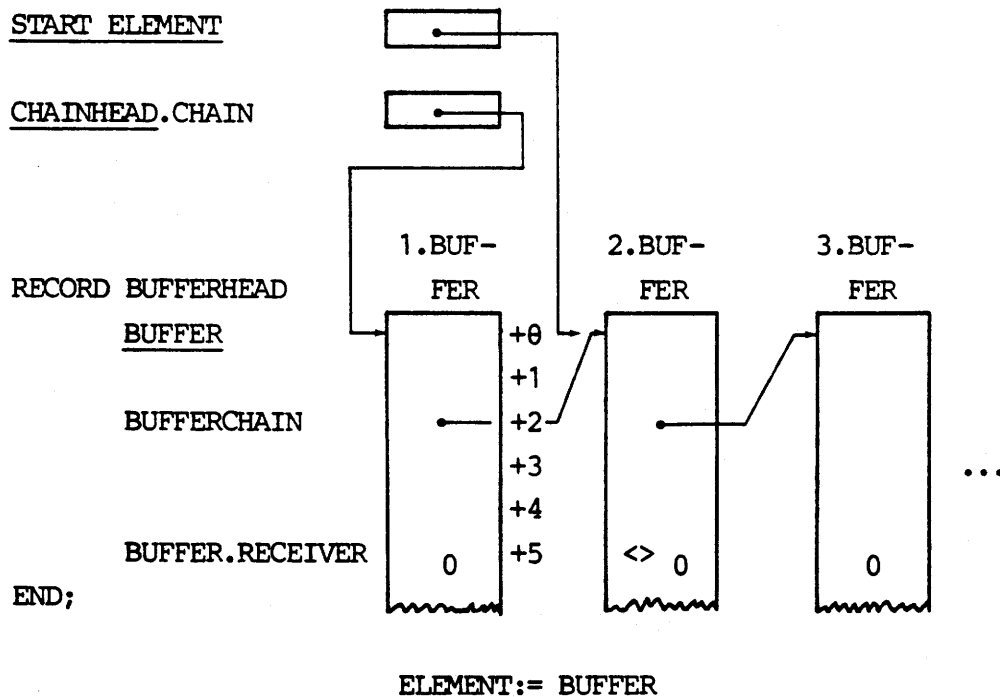
* When served request a fetch results in executing the current instruction (PC unchanged) once again with a probably changed set of registers if so specified in the microprogrammed instruction just interrupted.



X = DON'T CARE

	CALL:	RETURN:
; AC0	-	UNCHANGED
; AC1	-	UNCHANGED
; AC2	START ELEMENT	FREE ELEMENT
; AC3	-	UNCHANGED

This instruction searches a chain for a free element and delivers it if present, and a zero if not found. The chain-datastructure is illustrated in fig 5.9.



EXAMPLE: The MUS-buffer chain is searched for a free buffer starting from the one addressed in 'START ELEMENT'.

Figur 5.9

The instruction may be interrupted by interrupt and data channel request following the algorithm:

```

START: element:= start element           ; SFREE:
LOOP:  If element = 0 then goto EXIT
      Q:= word (element. receiver)
      If Q = 0 then goto EXIT           ; AC2:=
      element:= element.chain          ; Next element;

TEST:  If (INT REQ or DMA REQ) = 0
      then goto LOOP

WAIT:  Servereq (PC)                    ; Dcr. PC and serve req.

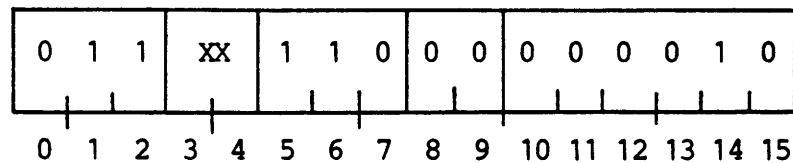
EXIT:  Fetchnext (PC)                   ; Incr. PC and exec instr.

```

5.10 Process Link

5.10

LINK



X = DON'T CARE

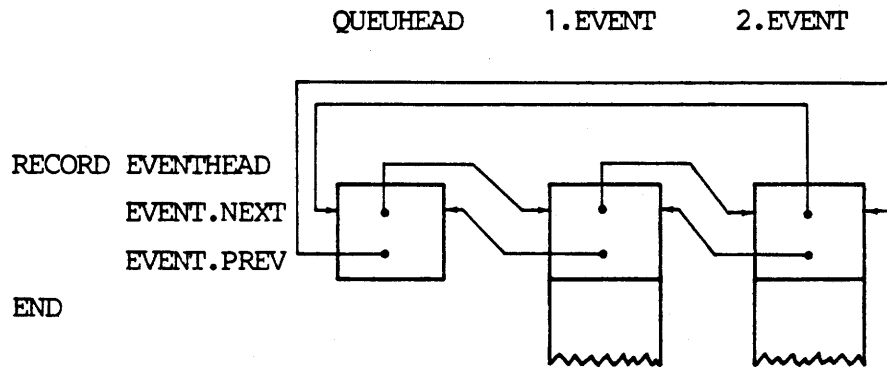
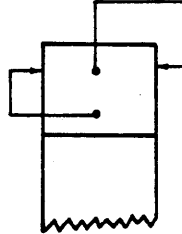
CALL:

; AC0	-	DESTROYED
; AC1	QUEUEHEAD	QUEUEHEAD
; AC2	NEW ELEMENT	NEW ELEMENT
; AC3	-	QUEUEHEAD

This instruction links an element to the end of a queue.
 A queue consists of one or more queue elements. One of the
 elements is the queue head as shown in fig 5.10.

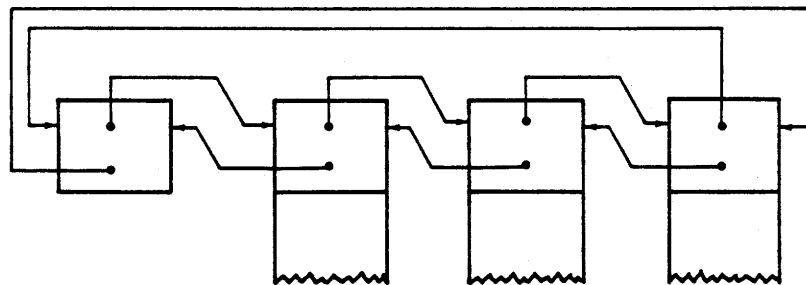
a Init:

NEW ELEMENT
 (neutral)



b Inserted:

QUEUEHEAD 1.EVENT 2.EVENT 3.EVENT
 (NEWELEMENT)



EXAMPLE: Bufferinsertion in the MUS-eventqueue of a process.

Figur 5.10: a+b

The instruction executes the following algorithm:

```

; LINK:
START: element:= new element      ; Update
      oldtail:= word (HEAD.prev)  ; Link element
      word (HEAD.prev):= element  ;
      word (element.next):= HEAD
      word (element.prev):= oldtail
      word (oldtail.next):= element

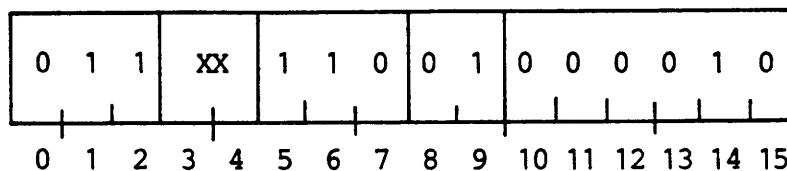
EXIT:  Fetchnext (PC)             ; Incr. PC and exec instr.

```

5.11 Process Remove

5.11

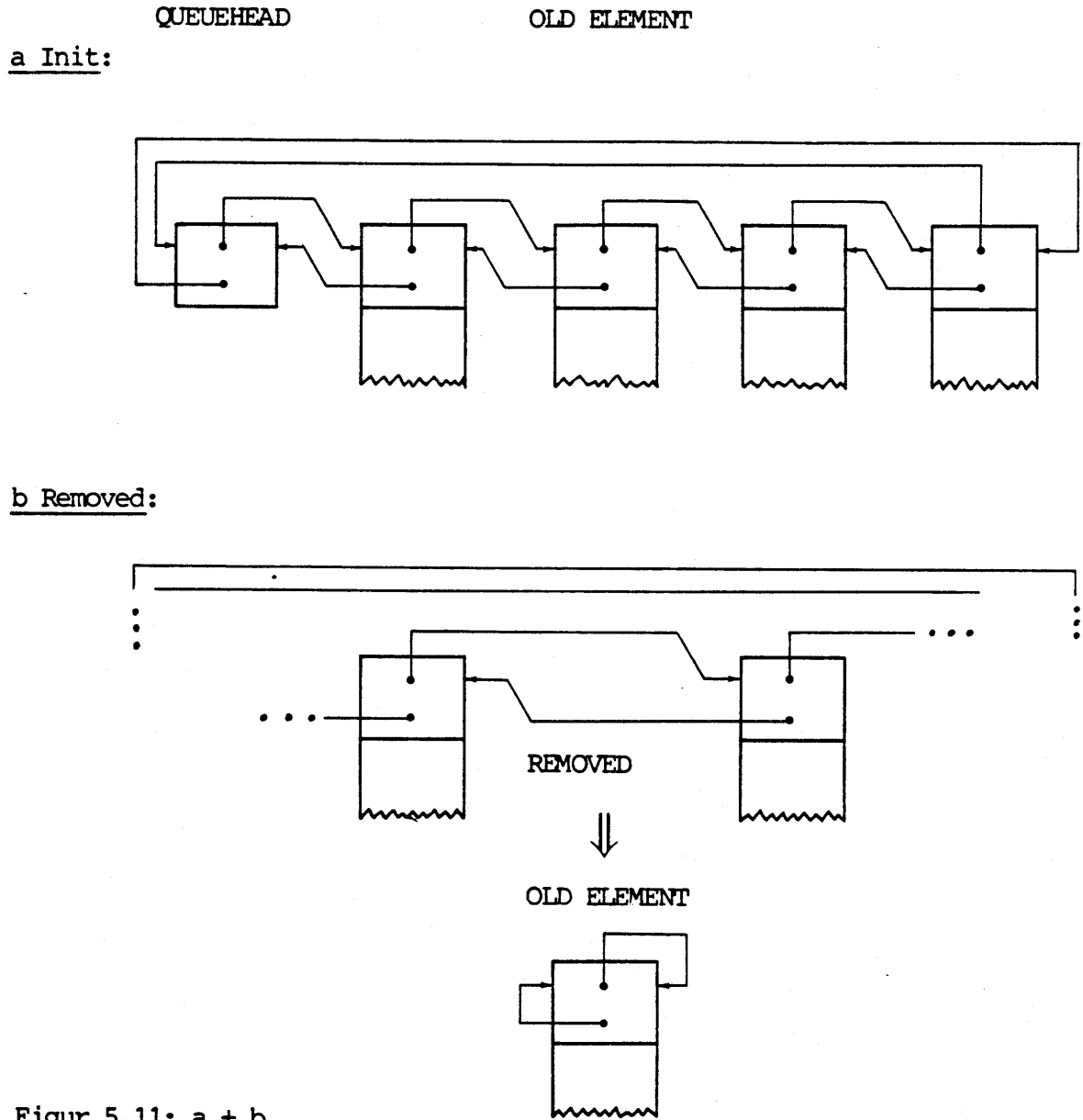
REMEL



X = DON'T CARE

	CALL:	RETURN:
; AC0	-	PREDECESSOR
; AC1	-	UNCHANGED
; AC2	OLD ELEMENT	OLD ELEMENT
; AC3	-	SUCCESSOR

This instruction removes an element from a queue as shown in fig. 5.11.



Figur 5.11: a + b

The instruction executes the following algorithm:

```

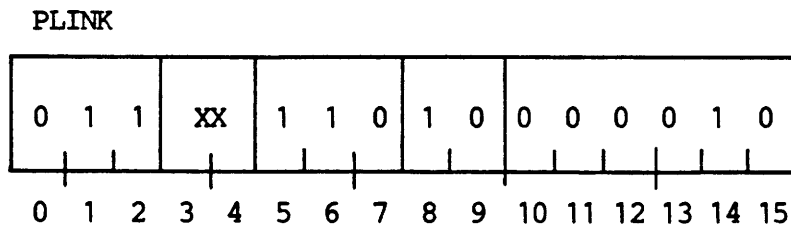
START: element:= old element           ; REMEL:
      successor: word (element.next)   ; Update queue:
      predecessor:= word (element.prev)
      word (predecessor.next):= successor
      word (successor.prev):= predecessor ; Remove element
      word (element.next):= element
      word (element.prev):= element

EXIT: Fetchnext (PC)                   ; Incr. PC and exec. instr.

```

5.12 Process Link Priority

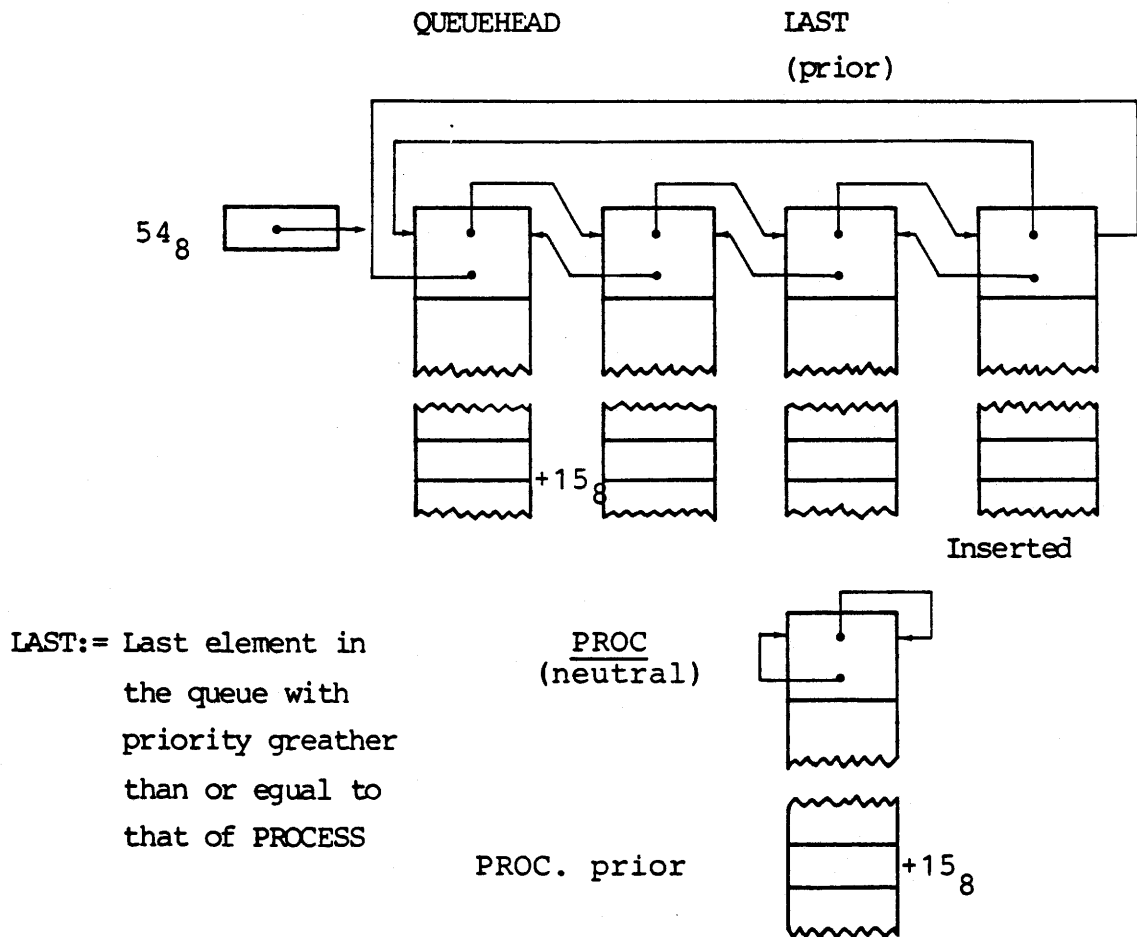
5.12



X = DON'T CARE

	CALL:	RETURN:
; AC0	-	DESTROYED
; AC1	-	QUEUE HEAD
; AC2	PROCESS	PROCESS
; AC3	-	QUEUE HEAD

This instruction links a process to the running queue as the last process among processes of same priority.



The **QUEUEHEAD** is itself an active element and points out first process in the running queue, that is - current process. If neutral - an empty queue - the first element (the head) points out itself: the dummy process.

Figur 5.12

The instruction may be interrupted by interrupt and data channel request following the algorithm:

```

                                ; PLINK:
START: word (PROC.state):= 0      ; Proc state:= runnig
      priority:= word (Proc.prior) ; ACO:= proc.priority
      HEAD:= word (54g)           ; HEAD:= running queue head
      element:= HEAD              ; AC3:= HEAD

LOOP:  element:= word (element.next) ; AC3:= next element
      Q:= word (element.prior)      ; AC1:= priority of next
      If Q < priority then goto EXIT

TEST:  If (INT REQ or DMA REQ) = 0
      then goto LOOP

WAIT:  Servereq (PC)              ; Dcr. PC and servereq
      Fetchnext (PC)              ; Incr. PC and exec instr.

EXIT:  predecessor:= word (element.prev) ; Update queue
      word (element.prev):= proc      ;
      word (proc.next):= element      ; insert dement.
      word (proc.prev):= predecessor  ;
      word (predecessor.next):= proc  ;

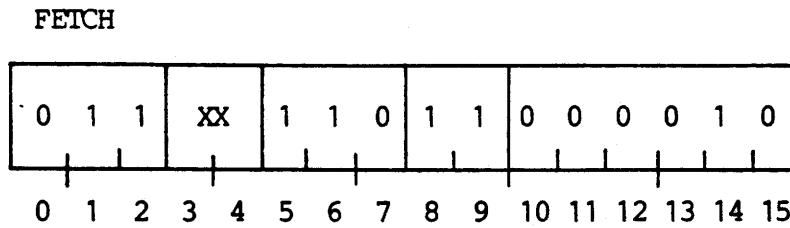
      Fetchnext (PC)              ; Incr. PC and exec.instr.

```


5.13

Instruction Fetch (Musil)

5.13

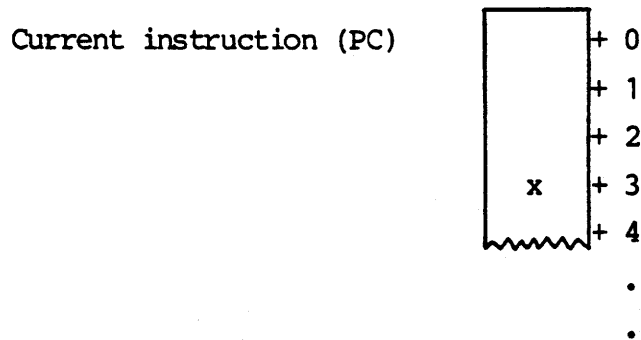


X = DON'T CARE

	CALL:	RETURN:
;AC0	-	DESTROYED
;AC1	-	DESTROYED
;AC2	CUR	CUR
;AC3	-	UNCHANGED

This instruction decodes MUSIL-instructions and performs a vector jump as shown in fig. 5.13. The MUSIL instruction-counter (denoted MPC) is found + 33₈ relative to current process description address CUR.

Instruction address table.

MPC: MUSIL program counter, word (CUR.33₈)DISP: (word(MPC) shift(-8)) and 377₈

EXAMPLE: DISP - vector jump to the MUSIL-instruction pointed out by x above (DISP = 3)

Figur 5.13

This instruction executes the following algorithm:

```

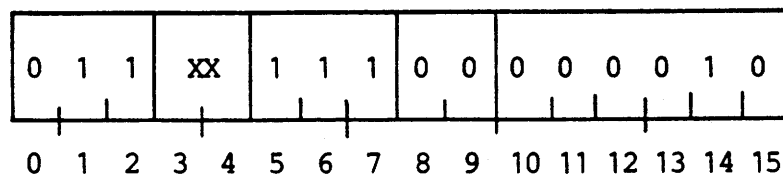
                                ; FETCH:
START:                          ; Fetch MPC:= word (CUR.33g)
                                ; Q = word (MPC) = next instruction;
    Q:= word MPC                ; Increment MPC
    Incr. MPC                   ; Decode instruction:
    Result:= Q and 337g
                                ; DISP:= word (MPC) (0:7)
    Q:= Q-result
    DISP:= Q shift (-8)
                                ; Modify PC
EXIT: PC:= word (PC + DISP)     ; Incr. PC and exec instr.
    Fetchnext (PC)

```

5.14 Take Address (Musil)

5.14

TKADD



X = DON'T CARE

	CALL:	RETURN:
;AC0	MODIFBITS	MODIFBITS SHIFT (-2)
;AC1	-	ADDRESS
;AC2	CUR	CUR
;AC3	-	DESTROYED

This instruction supplies the ADDR of an integer or string addressed by the MUSIL PC (MPC) and increments MPC.

```
Integer, MODIF(14:15) = 00: Addr := Word (MPC) ;
String,  - - -      = 01: -   - - -   ;
File,    - - -      = 10: -   - - -   ;
Mfield,* - - -      = 11: Addr := word (zone.zfirst) + Field (8:15),
                               zone = word (CUR.zn + Field (0:7))
```

*zonerecordfield

Field = word (MPC)

The instruction executes the following algorithm:

```

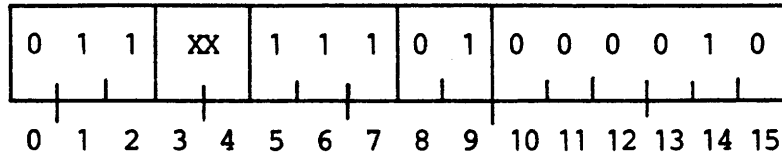
; TKADD:
START:                               ; Fetch MPC:= word (CUR.338)
    address:= word (MPC)              ; Update AC1;
    incr. MPC                          ; Increment MPC
    If modif (14:15) = 3 then
    begin
        Q:= address and 3778          ; Update AC1:
        address:= (address -Q) shift (-8)
        Q1:= address + cur             ; Q1:= Cur + address (0:7)
        Q1:= word (Q1.Zn)              ; Zone:= word (Q1.Zn)
        Q1:= word (Q1.zfirst)         ; Q1:= word (zone.zfirst)
        address:= Q1+Q                 ; Update AC1
    end                                 ; ADDRESS = Q1 + address (8:15)

EXIT: Modif:= modif shift (-2)        ; Update AC0: modif shift (-2);
    Fetchnext (PC)                    ; Incr. PC and exec. instr.
```

5.15 Takevalue (Musil)

5.15

TKVAL



X = DON'T CARE

	CALL:	RETURN:
;AC0	MODIFBITS	MODIFBITS SHIFT (-2)
;AC1	-	VAL
;AC2	CUR	CUR
;AC3	-	UNCHANGED

This instruction returns the value of an integer in MUSIL.

```

MODIF (14:15): 00 VAL = WORD (MPC)
- - : 01 VAL := R
- - : 10 VAL := WORD (WORD (MPC))
- - : 11 VAL := R

```

R= word (CUR.32_g), the interpreter register

This instruction is executed in following the algorithm:

```

; TKVAL:
START: If modif and (14:15) = 0 then
    begin
        VAL:= word (MPC)
        incr MPC
        goto EXIT
    end
    If modif (15) = 1 then
        begin
            VAL:= R
            goto EXIT
        end
        Q:= word (MPC)
        VAL:= word (Q)
        incr MPC
    end
EXIT: Modif:= modif shift (-2)
    Fetchnext (PC)

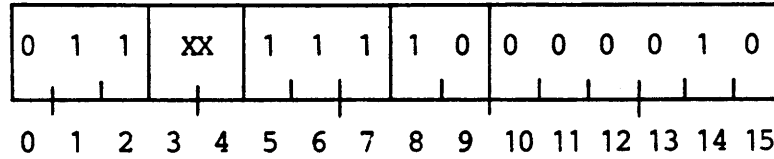
```

; Case modif 0:
; AC1:= value
; Increment MPC;
; Case modif 1 or 3:
; AC1:= value;
; Case modif 2 :
; AC1:= value
; Increment MPC;
; Update AC0;
; Incr. PC and exec. instr.

5.16 Compare Byte Strings

5.16

COMP



X = DON'T CARE

This instruction compares two byte strings and returns:
 RESULT = byte (STR ADDR1 + x) - byte (STR ADDR2 + x),
 if the strings differ in position x else zero.

	CALL:	RETURN:
;AC0	COUNT	RESULT (R<0, R=0, R>0)
;AC1	STR ADDR1	UNDEFINED
;AC2	STR ADDR2	UNDEFINED
;AC3	-	DESTROYED

6. Processor options

6.

The RC3803 CPU can be equipped with the following optional features: a Real Time Clock and a Teletype Controller.

6.1 Real Time Clock

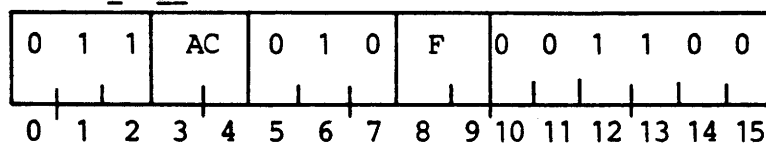
6.1

The Real Time Clock generates a continuous sequence of pulses independently of processor timing. The clock can be used primarily for low resolution timing as compared to processor speed, but it has a high long-term accuracy.

Following a power turn-on the various frequencies are only available after an interval of 5 seconds, because the crystal must be given this amount of time to settle down after excitation in order to emit a steady pulse train.

Selection of clock frequency is accomplished by means of the I/O instruction DATA OUT A, Real Time Clock:

DOA <f> ac,RTC



This instruction will select the clock frequency according to the values of bits 14 and 15 in the specified AC as listed below:

AC bits 14 & 15:	00	01	10	11
Frequency:	50 Hz	10 Hz	100 Hz	1000 Hz

In addition the instruction will cause the Busy and Done flags to be set according to the control code specified by "F" (cf. section 4.6). Setting the Busy flag by means of this instruction will allow the next pulse from the clock to set Done thus requesting an interrupt if the Interrupt On flag is 1.

The interrupt priority level of this device is associated with bit 13 of the interrupt priority mask.

The DATA OUT A instruction applied to select the clock frequency is needed only once. The first interrupt after this instruction has set Busy = 1 can come at any time up to the clock frequency, but once the first interrupt has appeared the following interrupts will adhere to the selected frequency - provided that the program sets Busy = 1 before the next interrupt is due. This is done by the instruction:

NIOS 14.

The I/O RESET instruction will - whether it appears in the program or is generated by using the Diagnostic Front Panel - reset the clock to a frequency of 50 Hz.

6.2 Teletype Controller

6.2

The Teletype Controller provides for two-way communication between the computer and the operator. The input device is the Teletype keyboard and the output device is the Teletype printer. All information exchanges between the computer and the keyboard/printer use a subset of the 128 character alphanumeric ASCII code as listed in Appendix B. In addition to a keyboard and a printer, some models of the Teletype terminal can be equipped with a paper tape reader/punch combination. Terminals so equipped are designated Automatic Send/Receive (ASR) terminals, while those not so equipped are designated Keyboard Send/Receive (KSR) terminals.

6.2.1 Instructions

6.2.1

Since the terminal is in effect two peripheral devices coupled together, the controller contains both an input buffer and an output buffer. These buffers are independent of one another and are both 8 bits in length.

Similarly two completely separate sets of Busy and Done flags are available for input and output operations respectively.

The Busy and Done flags are controlled by means of the two standard device flag commands in the instructions according to the following list:

"F" = S Sets Busy = 1 and Done = 0 and either reads a character into the input buffer or transfers a character in the output buffer to the printer (or the punch).

"F" = C Sets Busy = 0 and Done = 0 thereby stopping all data transfer operations. This command - if issued while a transfer is in process - will result in partial reception of the character code being transferred.

"F" = P No effect.

The instructions used to read the character buffer and to load the character buffer are the standard I/O instructions with the appropriate device codes. An extract of Appendix A containing these codes appear below:

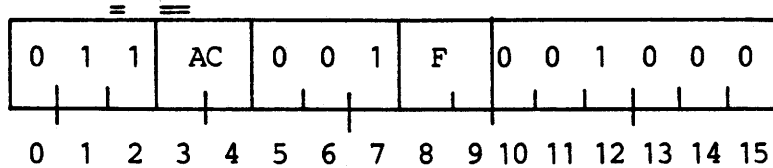
Octal

Code	Mnemonic	Maskbit	Device
10	TTI	14	Teletype input, first controller
11	TT0	15	Teletype output, first controller
50	TTI1	14	Teletype input, second controller
51	TT01	15	Teletype output, second controller

6.2.1.1 READ CHARACTER BUFFER

6.2.1.1

DIA <f> ac,TTI



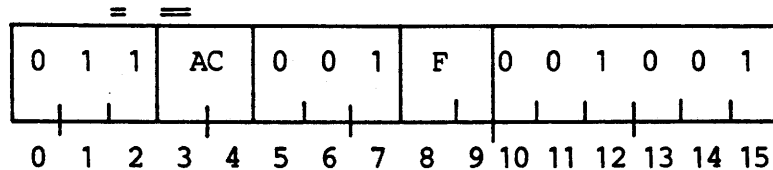
This instruction will place the contents of the input buffer in bits 8 to 15 of the AC specified in the instruction. Bit 8 is a parity check bit while bits 9 to 15 contain the character code proper. Bits 0 to 7 of the AC are all set to 0.

After the data transfer has been completed the controller's Busy and Done flags for input are set according to the control code specified by "F".

6.2.1.2 LOAD CHARACTER BUFFER

6.2.1.2

DOA <f> ac, TTO



This instruction will place bits 9 to 15 of the specified AC in the output buffer of the controller. After the transfer has been completed the controller's Busy and Done flags for output are set according to the control code specified by "F". The contents of the AC specified in the instruction will remain unaltered.

6.2.2 Programming

6.2.2

On account of the two-sided nature of the Teletype terminal this section will describe input and output procedures separately.

6.2.2.1 Input

6.2.2.1

Input operations - whether full- or half-duplex - do not have to be initialized by the program because the striking of a key on the keyboard automatically will transmit the corresponding character code to the controller. When the character has been assembled the input Busy flag is set to 0, the input Done flag is set to 1 and a program interrupt consequently requested - provided that the priority mask bit is 0.

The character can then be read by issuing the READ CHARACTER BUFFER instruction (DIA). The instruction should be issued with either a C or an S command so that the input Done flag is set to 0; this will allow the controller to initiate a further program interrupt request when the next character has been fully assembled.

6.2.2.2 Output

6.2.2.2

Output operations are initiated by the program using the LOAD CHARACTER BUFFER instruction (DOA). The instruction should be issued with an S command, which will set the Busy flag to 1 and allow the transmitting of the character to the terminal. When the transmission has been completed the output Busy flag is set to 0 and the output Done flag is set to 1 thus issuing a program interrupt request.

The output buffer must be reloaded by means of the LOAD CHARACTER BUFFER instruction every time a character is to be sent to the terminal. Thus to transmit a multi-character message a sequence of LOAD CHARACTER BUFFER instructions with S commands must be issued. The program must make allowance for complete transmission of every single character before transmission of the next character is initiated.

6.2.3 Programming Examples

6.2.3

The following examples show sections of programs which will handle character operations involving the Teletype keyboard, printer, paper tape reader, and paper tape punch.

Example 1 reads a character from the Teletype keyboard, example 2 reads a character from Tape reader, and example 3 prints a character on the Teletype printer and - if the tape punch on an ASR terminal is turned on - simultaneously punches the character on the tape.

6.2.3.1 Example 1

6.2.3.1

```

SKPDN   TTI       ;Character buffer loaded yet?
JMP     .-1      ;No
DIAC    1,TTI    ;Read character and clear Done
                    flag

```

6.2.3.2 Example 2

6.2.3.2

```

NIOS    TTI       ;Start reader
SKPDN   TTI       ;Frame buffer loaded yet?
JMP     .-1      ;No
DIAC    1,TTI    ;Read frame and clear Done flag

```

6.2.3.3 Example 3

6.2.3.3

```

SKPBZ   TIO       ;Printer free?
JMP     .-1      ;No, try again
DOAS    1,TIO     ;Print character

```

6.2.3.4 Example 4

6.2.3.

The subroutine shown in this example and called from the main program by a JUMP TO SUBROUTINE instruction (JSR to TTYRD) illustrates reading and echoing characters on the Teletype with Teletype interrupts disabled. AC0 is used to store the character.

```

TTYRD:  SKPDN    TTI      ;Has character been typed?
        JMP     .-1      ;No, then wait
        DIAC    0,TTI    ;Yes, then read character and
                        clear Done flag
        SKPBZ   TIO      ;Is TIO ready?
        JMP     .-1      ;No, then wait
        DCAS    0,TIO    ;Yes, then echo character
        JMP     0,3      ;Return

```

6.2.3.5 Example 5

6.2.3.5

This example shows how Teletype may be programmed using the program interrupt facility. To do so makes it possible to perform a number of calculations in the intervals of time between Teletype characters.

This routine will read a line and echo it on the Teletype printer using the interrupt priority system. The characters are read into a buffer area beginning at location 1000_g. The routine is terminated by either a carriage return character or line overflow. Line overflow is determined by the value of MAXLL (maximum line length).

```

        .LOC      0          ;
0                ;Program counter stored here when
                ;an interrupt occurs.

        IHAND     ;Address of interrupt handler
        .LOC      400       ;
START:  LDA      1,BUFFER  ;Set up buffer pointer in
                ;autoincrement location 23

        STA      1,23      ;
        LDA      1,MAXLL   ;Get maximum line length
        STA      1,CNTR    ;Initialize line overflow counter
        SUBZL    1,1       ;Set AC1 = 1
        DOBS     1,CPU     ;Mask out TIO and turn on
                ;interrupts

        .
        .
        .
HANG:   LDA      0,CNTR    ;When need full line to continue
                ;hang up here until reading is all
                ;done

        MOV      0,0,SZR   ;
        JMP      .-2       ;

        .
        .
        .
        .
BUFFR:  777                ;Buffer begins at location 1000
MAXLL:  110                ;Maximum of 7210
                ;characters per line
CNTR:   0                  ;Line overflow counter

        .
        .
        .
IHAND:  SKPDN    TTI       ;Make sure TTI caused the
                ;interrupt

```

```

HALT                ;Error - some other peripheral
                    ;interrupted

STA      0,SAV0     ;Save accumulators that will be
                    ;used

STA      1,SAV1     ;

DIAC     0,TTI      ;Read character and clear Done

STA      0, 23      ;Store character in buffer

SKPBZ    TTO        ;Make sure TTO not busy

JMP      .-1        ;

DOAS     0,TTO      ;Echo character

LDA      1,CR       ;Is it a carriage return?

SUB      0,1,SZR    ;

JMP      .+4        ;No

SUBC     0,0        ;Yes, clear ACO without changing
                    ;carry

STA      0,CNTR     ;Zero out CNTR to indicate line
                    ;done

JMP      .+3        ;

DSZ      CNTR       ;If not a carriage return,
                    ;decrement CNTR

JMP      OUT        ;Line not yet done, go dismiss

LDA      0,TIMSK    ;Line is done

MSKO     0          ;Mask out TTI (and TTO) to inhibit
                    ;further input

OUT:     LDA      0,SAV0 ;Restore accumulators
        LDA      0,SAV1 ;
        INTEN    ;Turn interrupts back on
        JMP      0      ;Return to interrupted program

SAV0:    0
SAV1:    0
CR:      215
TIMSK:   3

```


7. Program Loading

7.

7.1 Introduction

7.1

Whenever the computer is used for information processing of any kind the program must - as previously mentioned - reside in main memory. But to read a program into memory is in itself a kind of information processing and therefore requires the existence in memory of a program - called a loading program - to perform this duty.

Although the loading program will normally be present, it may from time to time be necessary to read it into memory. This is done by a small, specialized loading program which is called a "bootstrap loader" and whose only function is to read into memory the more general-purpose loading program.

Two methods are available for entering the bootstrap loader into memory. One is for the operator to enter it manually utilizing the data switches and the deposit switch on the Diagnostic Front Panel. The other is to use the Automatic Program Load option if the computer in question is so equipped.

In this chapter only automatic program loading is described. For details about manual loading the reader must consult the Reference Manual for the Diagnostic Front Panel - RCSL: 52-AA542.

7.2 Automatic Loading

7.2

To use the Automatic Program Load option, the operator must first select the input device and set up the loading program on this device in preparation to be read. In addition the device code of this unit must be set up in its binary form on the data switches 10 to 15 on the front frame of the CPU board (cf. the illustration appearing in the following chapter). The setting of data switch 0 on the front panel depends on the type of input device selected. If this is a data channel device - for instance magnetic tape - data switch 0 must be set to 1. If it is a low-speed device - for instance a paper tape reader - data switch 0 must be set to 0.

When this has been done, push the AUTOLOAD switch on the operator panel. This will cause the bootstrap loader to be read, deposited in memory locations 0 to 37_8 and started location 0. The bootstrap loader will then read the data switches (0 and 10 to 15), set up its own I/O instructions with the device code as read and finally perform a program load procedure which depends on the setting of data switch 0.

If data switch 0 has been set to 1, the bootstrap loader will start the device for data channel transfer starting storage at location 0 and will then loop at location 377_8 until a data channel transfer places a word in this location. When this happens, the word placed in this location is executed as an instruction; typically this will be a JUMP into the data which have been placed in locations 0 to 376_8 .

NOTE: For proper program loading via the data channel the device in use must be initialized for the reading operation by an I/O RESET instruction followed by a NIOS instruction. Furthermore the device must stop reading when 256_{10} words has been read; otherwise the available memory locations will overflow.

If data switch 0 has been set to 0, the bootstrap loader will read the loading program via programmed I/O. The device must supply data as 8-bit bytes; each pair of bytes read will be memory stored in as a single word wherein the first and second byte will become respectively the left and right halves of the word. To simplify the positioning of the input medium - for instance paper tape - the bootstrap loader will ignore leading null characters, i.e. it will not store any word until it has read a non-zero synchronization byte.

The first word following this synchronization byte must be the negative of the total number of words to be read including this first word. The number of words to be read - including the first - cannot exceed 192_{10} . The bootstrap loader will store the words read in memory starting in location 100_8 . When the last word has been read the bootstrap loader will transfer control to that location.

The Automatic Loading hardware in RC3803 is capable of containing up to 16 times 32 word programs, one of this programs is listed on the following pages, a bootstrap loader capable of loading in either of the manners described above.

A list of the available bootstrap loaders in the Automatic Program Load option, F10A is too shown.

For details about the RC3803 program load refer to:

GENERAL INFORMATION

Hardware Testprograms and Program Load to RC3803

RCSL - 52AA894

BOOTSTRAP LOADER FOR
AUTOMATIC PROGRAM LOAD

```

00000 060477 BEG:  READS  0      ;READ SWITCHES INTO AC0
00001 105120      MOVZL  0,1    ;ISOLATE DEVICE CODE
00002 124240      COMOR   1,1    ;-DEVICE CODE  -1

00003 010011 LOOP: ISZ     OP1    ;COUNT DEVICE CONTROL INTO
                                ALL
00004 010031      ISZ     OP2    ;I0 INSTRUCTIONS
00005 010033      ISZ     OP3    ;
00006 010014      ISZ     OP4    ;
00007 125404      INC     1,1,SZR ;DONE?
00010 000003      JMP     LOOP   ;NO INCREMENT AGAIN

00011 060077 OP1:  060077      ;START DEVICE;(NIOS 0) -1
00012 030017      LDA     2,C377  ;YES,PUTJMP 377INTO
                                LOCATION 377
00013 050377      STA     2,377   ;
00014 063377 OP4:  063377      ;BUSY ? :( SKPEN 0 ) -1
00015 000011      JMP     OP1    ;NO, GO TO OP1
00016 101102      MOVL   0,0,SZC  ;LOW SPEED DEVICE?(TEST
                                SWITCH 0)
00017 000377 C377: JMP     377    ;NO, GO TO 377 AND WAIT
                                FOR CHAN.

00020 004031 LOOP2:JSR    GET+1   ;GET A FRAME
00021 101065      MOVCL  0,0.SNR  ;IS IT NONZERO?
00022 000020      JMP     LOOP2   ;NO, IGNORE AND GET ANOTHER

00023 004030 LOOP4:JSR    GET     ;YES, GET A FULL WORD
00024 046027      STA     1,@C77  ;STORE STARTING AT 100
00025 010100      ISZ     100    ;COUNT WORD - DONE?
00026 000023      JMP     LOOP4   ;NO, GET ANOTHER
00027 000077 C77:  JMP     77     ;YES - LOCATION COUNTER AND
                                JUMP TO LAST WORD

00030 126420 GET:  SUBZ   1,1     ;CLEAR AC1, SET CARRY
                                OP2:

```

Figur 7.1

```
00031 063577 LOOP3:063577      ;DONE ? : ( SKPDN 0)-1
00032 000031      JMP      LOOP3      ;NO, WAIT
00033 060477 OP3: 060477      ;YES, READ INTO AC0:(DIAS
                                0,0) -1
00034 107363      ADDCS   0,1,SNC      ;ADD 2 FRAMES SWAPPED-
                                GOTSECOND?

00035 000031      JMP      LOOP3      ;NO, GO BACK AFTER IT.
00036 125300      MOVS    1,1      ;YES, SWAP AC1
00037 001400      JMP      0,3      ;RETURN WITH FULL WORD
```

LIST OF AVAILABLE
PROGRAM LOADS in F10B

DEVICE NO. (OCTAL)	BIT 0	AUTOLOAD	
		NO.	PROGRAM MODULE
0	x	0	CONSOLE INITIALIZATION (BAUD RATE, NO. OF STOP BITS AND MEMORY RESET)
1	x	1	MEMORY TESTPROGRAM
2	0	2	CONSOLE ECHO PROGRAM
2	1	2	CONSOLE CHARACTER GENERATOR
3-15 17 21-55 57-60 62-72 74-77	0	3	STANDARD AUTOLOAD LOW SPEED DEVICE (i.e. Ptr Dev. 12 _g)
3-15 17 21-55 57-60 62-72 74-77	1	3	STANDARD AUTOLOAD DATA CHANNEL PROGRAM LOAD Mag. tape Dev. 30 _g) FPA Dev. 46, Dev. 74)
16	x	4	CARD READER PROGRAM LOAD (CDR)
56	x	4	CARD READER PROGRAM LOAD (CDR)
61	0	5	FLEXIBLE DISC PROGRAM LOAD (FDD)
61	1	5	NO FUNCTION
73	0	6	DISC PROGRAM LOAD (DKP) (incl. a Disc recalibration)
73	1	6	DISC PROGRAM LOAD (DKP) (no recalibration)
20	0	7	Disc Storage Module PROGRAM LOAD

RELATION BETWEEN AUTOLOAD DEVICE NO
(SET ON THE FRONT PANEL OF RC3803)
AND THE SELECTED PROGRAM MODULE

Figur 7.2

8. Switches and Indicators 8.

This chapter contains a description of the switches and indicators placed on the front frame of the CPU board. An illustration of the front panel is found at the extreme end of the chapter.

8.1 Switches 8.1

Four groups of switches are placed on the front panel, namely the ENABLE TCP switch, the AUTOLOAD DEVICE SELECT switches, the PARITY ERROR switches, and the MEMORY EXTENSION SELECT switch.

8.1.1 ENABLE TCP 8.1.1

This switch transfers control to and from the Diagnostic Front Panel, details of which can be found in Reference Manual for the Diagnostic Front Panel - RCSL: 52-AA542.

When this switch is in the UP position, the Diagnostic Front Panel can be connected to or disconnected from the CPU without creating any disturbance for CPU program execution. Furthermore the AUTOLOAD DEVICE SELECT switches are operative when ENABLE TCP is in this position.

Whenever the Diagnostic Front Panel is not connected to the CPU, the ENABLE TCP switch is inoperative, i.e. pushing this switch will not affect the CPU.

When the ENABLE TCP switch is in the DOWN position all control of the CPU is carried out from the Diagnostic Front Panel connected to the CPU.

NOTE: The ENABLE TCP switch must be in the UP position before the Diagnostic Front Panel is connected or disconnected to the CPU.

8.1.2 AUTOLOAD DEVICE SELECT

8.1.2

These switches are operative when the ENABLE TCP switch is in the UP position as mentioned above. They are used for external, manual setting of specific bits of a word, the bits in question being bit 0 and bits 10 to 15.

Setting these switches is imperative in connection with the use of the Automatic Program Loading feature as outlined in the previous chapter. In this case the switches 10 to 15 are set according to the binary code of the input device being used, whereas switch 0 is used to distinguish between the types of device available, i.e. whether the device is a data channel device or a programmed I/O device.

Apart from this the switches can be used in conjunction with normal program operation by including the instruction READ SWITCHES; this instruction will - as explained in section 4.7.3 - place the bit values indicated by these switches in their respective positions in an accumulator specified by the instruction. When loaded into the accumulator the bit setting indicated will be accessible to the program. When the bits are loaded into the accumulator bits 1 to 9 will be read as logic zeroes.

8.1.3 PARITY ERROR

8.1.3

This group contains two switches: STOP and RESET.

When the STOP switch is in the DOWN position a parity error detected during a memory read cycle will cause the CPU to suspend processing in the microprogram. This will allow connecting of the Diagnostic Front Panel to the CPU while the CPU is still at that point of execution where the error was registered. Thus information about the memory address giving rise to the parity error can be read out from the memory address register so that corrective action can be decided upon.

To restart the CPU following a parity error - if so desired - is accomplished either by pushing the STOP switch to the UP position or by pushing the RESET switch to the DOWN position.

When the STOP switch is in the UP position the detection of a parity error will be indicated (cf. section 8.2.1), but processing will continue without interruption.

When the RESET switch is pushed to the DOWN position the parity error indicators (cf. section 8.2.1) will be reset; if the CPU has suspended processing following the detection of a parity error, this action will simultaneously restart the CPU.

CAUTION

If the switch AUTO is pushed while the RESET switch is still in the DOWN position, the CPU will restart in the address determined by the positions of the AUTOLOAD DEVICE SELECT switches - direct if switch 0 is set to 0, indirect if switch 0 is set to 1.

(A description of the AUTO switch mentioned above is not included in this manual. This switch is a feature of the Diagnostic Front Panel and the external Autoload Panel; more detailed information must be sought in the relevant manuals.)

NOTE: Activating the RESET switch will only reset the indicators. The parity error causing the indication will still be present in the particular memory location. Only a write operation into that location will remove the error.

8.1.4 MEMORY EXTENSION SELECT

8.1.4

When this switch is in the DOWN position the Memory Extension feature is inoperative. If the switch is in the UP position the programmer can utilize the extended block of core memory by including the proper instructions in the program. (Refer to section 5.3.)

8.2 Indicators

8.2

Two groups of indicators are placed on the front panel, namely the PARITY ERROR indicators and the CPU-STATUS indicators.

8.2.1 PARITY ERROR

8.2.1

This group consists of two indicating lights: LEFT and RIGHT. The LEFT indicator is lit whenever a parity error is detected in the left byte (bits 0 to 7) of a word being read during a memory read cycle.

The RIGHT indicator is lit whenever a parity error is detected in the right byte (bits 8 to 15) of a word being read during a memory read cycle.

The indicators - either or both - can only be cleared by pushing the RESET switch as previously described.

8.2.2 CPU-STATUS

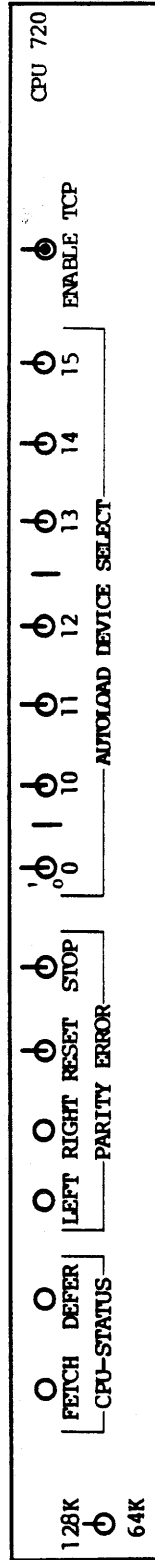
8.2.2

This group consists of two indicating lights: FETCH and DEFER.

The FETCH indicator is lit whenever the CPU is reading an instruction from core memory.

The DEFER indicator is lit whenever the next microcycle will be used to follow an indirect addressing chain.

FRONT FRAME OF CPU BOARD



A. I/O Device Codes and Mnemonic

A.

Decimal code	Octal code	Mnemonic	Maskbit	Device
01	01			Extended Memory
02	02			
03	03			
04	04			
05	05	ASL		Automatic System Load
06	06			
07	07			
08	10	TTI	14	Teletype Input
09	11	TTO	15	Teletype Output
10	12	PTR	11	Paper Tape Reader
11	13	PTP	13	Paper Tape Punch
12	14	RTC	13	Real Time Clock
13	15	PLT	12	Incremental Plotter
		SPC2	9	Third Standard Parallel Controller
14	16	CDR	10	Card Reader
15	17	LPT	12	Line Printer
16	20	DSC	4	Disc Storage Channel
17	21	SPC	9	Standard Parallel Controller
18	22	SPC1	9	Second Standard Parallel Controller
		ACU1		Second Dial-up Controller
19	23	PTR1	11	Second Paper Tape Reader
20	24	AMX3	2	Fourth 8 Channel Asynchronous Multiplexor
		TMX10	0	{ Second 64 Channel
21	25	TMX11	1	
22	26	TMX0	0	{ 64 Channel Asynchronous
23	27	TMX1	1	
24	30	MT	5	Magnetic Tape
25	31	PTP1	13	Second Paper Tape Punch
26	32	TTI2	14	Third Teletype Input
				OCP-Function Button Out
		IBM		First IBM Channel, Receiver Part

Decimal code	Octal code	Mnemonic	Maskbit	Device
27	33	TTO2	15	Third Teletype Output OCP-Function Button In
		IBM		First IBM Channel, Transmitter Part
28	34	TTI3	14	Fourth Teletype Input OCP-Numeric Keyboard In
29	35	TTO3	15	Fourth Teletype Output
		DISP	7	OCP-Display
30	36			OCP-Autoload
31	37	LPS	12	Serial Printer
32	40	REC	8	BSC Controller
33	41	XMT		
34	42	REC1	8	Second BSC Controller
35	43	XMT1		
36	44	MT1	5	Second Magnetic Tape
37	45	CLP	12	Charaband Printer
38	46	FPAR	3	Inter Processor Channel Receiver
39	47	FPAX	3	Inter Processor Channel Transmitter
40	50	TTI1	14	Second Teletype Input
41	51	TTO1	15	Second Teletype Output
42	52	AMX	2	8 Channel Asynchronous Multiplexor
43	53	AMX1	2	Second 8 Ch. Asynchronous Multipl.
44	54	HLC	8	HDLC Controller
		FPAR2	3	Third Inter Processor Ch. Receiver
45	55	HLC1	8	Second HDLC Controller
		FPAX2	3	Third Inter Processor Channel Transmitter
46	56	CDR1	10	Second Card Reader
47	57	LPT1	12	Second Line Printer
		LPS2	12	Third Serial Printer
48	60	SMX		Synchronous Multiplexor
49	61	FDD	7	Flexible Disc Drive
50	62	CRP	10	Card Reader Punch
		IBM1		Second IBM Channel, Receiver Part
51	63	CLP1	12	Second Charaband Printer
		IBM1		Second IBM Channel, Transmit. Part
52	64	FDD1	7	Second Flexible Disc Drive

Decimal Octal

code	code	Mnemonic	Maskbit	Device
53	65	LPS3	12	Fourth Serial Printer
		CLP2		Third Charaband Printer
54	66	DTC	9	Digital Cartridge Controller
		LPS4	12	Fifth Serial Printer
55	67	LPS1	12	Second Serial Printer
		CLP3		Fourth Charaband Printer
56	70	DST		Digital Sense
57	71	DOT		Digital Output
58	72	CNT		Digital Counter
		ACU		Dial-up Controller
59	73	DKP	7	Moving Head Disc Channel
60	74	FPAR1	3	Second Inter Processor Channel Receiver
61	75	FPAX1	3	Second Inter Processor Channel Transmitter
62	76	AMX2	2	Third 8 Channel Asynchronous Multiplexor
63	77	CPU		Central Processor

B. ASCII Character Codes

B.

Deci- mal	Octal	Hex	ASCII Cha- racter	Control Function	To Produce			Even Parity 8-bit code
					On TTY Mod 33,35	Cntr Shift	Char	
0	000	00	NUL	Null	+	+	P	00
1	001	01	SOH	Start of Heading	+		A	81
2	002	02	STX	Start of Text	+		B	82
3	003	03	ETX	End of Text	+		C	03
4	004	04	EOT	End of Transmission	+		D	84
5	005	05	ENQ	Enquiry	+		E	05
6	006	06	ACK	Acknowledge	+		F	06
7	007	07	BEL	Bell	+		G	87
8	010	08	BS	Backspace	+		H	88
9	011	09	HT	Horizontal Tap	+		I	09
10	012	0A	NL	New Line			line feed	0A
					+		J	0A*
					+		line feed	8A
11	013	0B	VT	Vertical Tab	+		K	8B
12	014	0C	FF	Form Feed	+		L	0C
13	015	0D	RT	Return			return	8D
					+		M	8D*
					+		return	0D
14	016	0E	SO	Shift Out	+		N	8E
15	017	0F	SI	Shift In	+		O	0F
16	020	10	DLE	Data Link Escape	+		P	90
17	021	11	DC1	Device Control 1	+		Q	11
18	022	12	DC2	Device Control 2	+		R	12
19	023	13	DC3	Device Control 3	+		S	93

* on even parity Teletypes these codes have odd parity

Deci- mal	Octal	Hex	ASCII Cha- racter	Control Function	To Produce On TTY Mod 33,35		Even Parity 8-bit code	
					Cntr	Shift Char		
20	024	14	DC4	Device Control 4	+	T	14	
21	025	15	NAK	Negative Acknow- ledge	+	U	95	
22	026	16	SYN	Synchronous Idle	+	V	96	
23	027	17	ETB	End Transmission Block	+	W	17	
24	030	15	CAN	Cancel	+	X	18	
25	031	19	EM	End of Medium	+	Y	99	
26	032	1A	SUB	Substitute	+	Z	9A	
27	033	1B	ESC	Escape		esc	1B	
					+	+	K	1B
28	034	1C	FS	File Separator	+	+	L	9C
29	035	1D	GS	Group Separator	+	+	M	1D
30	036	1E	RS	Record Separator	+	+	N	1E
31	037	1F	US	Unit Separator	+	+	O	9F
32	040	20	SP	Space			space	A0
33	041	21	!			+	1	21
34	042	22	"			+	2	22
35	043	23	#			+	3	A3
36	044	25	<			+	4	24
37	045	25	%			+	5	A5
38	046	26	&			+	6	A6
39	047	27	'			+	7	27
40	050	28	(+	8	28
41	051	29)			+	9	A9
42	052	2A	*			+	:	AA
43	053	2B	+			+	;	2B
44	054	2C	,				,	2C

Deci- mal	Octal	Hex	ASCII Cha- racter	Control Function	To Produce On TTY Mod 33,35 Cntr Shift Char	Even Parity 8-bit code
45	055	2D	-		-	2D
46	056	2E	.		.	2E
47	057	2F	/		/	AF
48	060	30	0		0	30
49	061	31	1		1	B1
50	062	32	2		2	B2
51	063	33	3		3	33
52	064	34	4		4	B4
53	065	35	5		5	35
54	066	36	6		6	36
55	067	37	7		7	B7
56	070	38	8		8	B8
57	071	39	9		9	39
58	072	3A	:		:	3A
59	073	3B	;		;	BB
60	074	3C	<		+ ,	36
61	075	3D	=		+ -	BD
62	076	3E	>		+ .	BE
63	077	3F	?		+ /	3F
64	100	40	@		+ P	C0
65	101	41	A		A	41
66	102	42	B		B	42
67	103	43	C		C	43
68	104	44	D		D	44
69	105	45	E		E	C5

Deci- mal	Octal	Hex	ASCII Cha- racter	Control Function	To Produce On TTY Mod 33,35 Cntr Shift Char	Even Parity 8-bit code
70	106	46	F		F	C6
71	107	47	G		G	47
72	110	48	H		H	48
73	111	49	I		I	C9
74	112	4A	J		J	CA
75	113	4B	K		K	4B
76	114	4C	L		L	CC
77	115	4D	M		M	4D
78	116	4E	N		N	4E
79	117	4F	O		O	CF
80	120	50	P		P	50
81	121	51	Q		Q	D1
82	122	52	R		R	D2
83	123	53	S		S	53
84	124	54	T		T	D4
85	125	55	U		U	55
86	126	56	V		V	56
87	127	57	W		W	D7
88	130	58	X		X	D8
89	131	59	Y		Y	59
90	132	5A	Z		Z	5A
91	133	5B	[+ K	DB
92	134	5C	\		+ L	5C
93	135	5D]		+ M	DD
94	136	5E	^		+ N	DE

Deci- mal	Octal	Hex	ASCII Cha- racter	Control Function	To Produce On TTY Mod 33,35 Cntr Shift Char	Even Parity 8-bit code
95	137	5F	-		+ 0	5F
96	140	60	\			60
97	141	61	a			E1
98	142	62	b			E2
99	143	63	c			63
100	144	64	d			E4
101	145	65	e			65
102	146	66	f			66
103	147	67	g			E7
104	150	68	h			E8
105	151	69	i			69
106	152	6A	j			6A
107	153	6B	k			EB
108	154	6C	l			6C
109	155	6D	m			ED
110	156	6E	n			EE
111	157	6F	o			6F
112	160	70	p			F0
113	161	71	q			71
114	162	72	r			72
115	163	73	s			F3
116	164	74	t			74
117	165	75	u			F5
118	166	76	v			F6
119	167	77	w			77

Deci- mal	Octal	Hex	ASCII Cha- racter	Control Function	To Produce On TTY Mod 33,35 Cntr Shift Char	Even Parity 8-bit code
120	170	78	x			78
121	171	79	y			F9
122	172	7A	z			FA
123	173	7B	{			7B
124	174	7C	:			FC
125	175	7D	}			7D
126	176	7E	~			7E
127	177	7F	DEL		rubout	FF

C. Double Precision Arithmetic

C.

A double length number consists of two words concatenated into a 32-bit string wherein bit 0 is the sign and bits 1-31 are the magnitude in two's complement notation. The high-order part of a negative number is therefore in one's complement form unless the low-order part is null (at the right only 0's are null regardless of sign). Hence, in processing double length numbers, two's complement operations are usually confined to the low-order parts, whereas one's complement operations are generally required for the high-order parts.

Suppose we wish to negate the double length number whose high and low-order words respectively are in AC0 and AC1. We negate the low-order part, but we simply complement the high-order part unless the low order part is zero. Hence

```

NEG      1,1,SNR
NEG      0,0,SKP ;LOW ORDER ZERO
COM      0,0      ;LOW ORDER NON-ZERO

```

Note that the magnitude parts of the sequence of negative numbers from the most negative toward zero are the positive numbers from zero upward. In other words, the negative representation $-x$ is the sum of x and the most negative number. Hence, in multiple precision arithmetic, low-order words can be treated simply as positive numbers. In unsigned addition a carry indicates that the low-order result is just too large and the high-order part must be increased. We add the number in AC2 and AC3 to the number in AC0 and AC1.

```

ADDZ     3,1,SZC
INC      0,0
ADD      2,0

```

In two's complement subtraction a carry should occur unless the subtrahend is too large. We could increment as in addition, but since incrementing in the high-order part is precisely the difference between a one's complement and a two's complement, we can always manage with only two instructions. We subtract the number in AC2 and AC3 from that in AC0 and AC1.

```
SUBZ    3,1,SZC
SUB     2,0,SKP
ADC     2,0
```

D. Instruction Use, Examples

D.

On the following pages are examples of how the instruction set of the RC3803 computer may be used to perform some common functions.

1. Clear an AC and the carry bit.

```
SUBO    AC,AC
```

2. Clear an AC and preserve the carry bit.

```
SUBC    AC,AC
```

3. Generate the indicated constants.

```
SUBZL   AC,AC           ;GENERATE +1
ADC     AC,AC           ;GENERATE -1
ADCZL   AC,AC           ;GENERATE -2
```

4. Let ACX be any accumulator whose contents are zero. Generate the indicated constants in ACX.

```
INCZL   ACX,ACX        ;GENERATE +2
INCOL   ACX,ACX        ;GENERATE +3
INCS    ACX,ACX        ;GENERATE +4008
```

5. Subtract 1 from an accumulator without using a constant from memory.

```
NEG     AC,AC
COM     AC,AC
```

6. Check if both bytes in an accumulator are equal.

```
MOVS    ACS,ACD
SUB     ACS,ACD,SZR
JMP     —           ;NOT EQUAL
—      —           ;EQUAL
```

7. Check if two accumulators are both zero.

```

MOV      ACS,ACS,SNR
SUB#     ACS,ACD,SZR
JMP      —           ;NOT BOTH ZERO
—       —           ;BOTH ZERO

```

8. Check an ASCII character to make sure it is a decimal digit. The character is in ACS and is not destroyed by the test. Accumulators ACX and ACY are destroyed.

```

LDA      ACX,C60      ;ACX=ASCII ZERO
LDA      ACY,C71      ;ACY=ASCII NINE
ADCZ#    ACY,ACS,SNC  ;SKIPS IF (ACS) > 9
ADCZ#    ACS,ACX,SZC  ;SKIPS IF (ACS) = 0

JMP      —           ;NOT DIGIT
—       —           ;DIGIT

C60:     60           ;ASCII ZERO
C71      71           ;ASCII NINE

```

9. Test an accumulator for zero.

```

MOV      AC,AC,SZR
JMP      —           ;NOT ZERO
—       —           ;ZERO

```

10. Test an accumulator for -1.

```

COM#     AC,AC,SZR
JMP      —           ;NOT -1
—       —           ;-1

```

11. Test an accumulator for 2 or greater.

```

MOVZR#   AC,AC,SNR
JMP      —           ;LESS THAN 2
—       —           ;2 OR GREATER

```


12. Assume it is known that AC contains 0, 1, 2, or 3.
Find out which one.

```

MOVZR#   AC,AC,SEZ
JMP      THREE           ;WAS 3
MOV      AC,AC,SNR
JMP      ZERO            ;WAS 0
MOVZR#   AC,AC,SZR
JMP      TWO              ;WAS 2
——      ——             ;WAS 1

```

13. Multiply an AC by the indicated value.

```

MOV      ACX,ACX         ;MULTIPLY BY 1
MOVZL    ACX,ACX         ;MULTIPLY BY 2
MOVZL    ACX,ACY         ;MULTIPLY BY 3
ADD      ACY,ACX
ADDZL    ACX,ACX         ;MULTIPLY BY 4
MOV      ACX,ACY         ;MULTIPLY BY 5
ADDZL    ACX,ACX
ADD      ACY,ACX
MOVZL    ACX,ACY         ;MULTIPLY BY 6
ADDZL    ACY,ACX
MOVZL    ACX,ACY         ;MULTIPLY BY 7
ADDZL    ACY,ACY
SUB      ACX,ACY         ;IN ACY
ADDZL    ACX,ACX         ;MULTIPLY BY 8
MOVZL    ACX,ACX
MOVZL    ACX,ACY         ;MULTIPLY BY 9
ADDZL    ACY,ACY
ADD      ACY,ACX
MOV      ACX,ACY         ;MULTIPLY BY 1010
ADDZL    ACX,ACX
ADDZL    ACY,ACX
MOVZL    ACX,ACY         ;MULTIPLY BY 1210
ADDZL    ACY,ACX
MOVZL    ACX,ACX
MOVZL    ACX,ACY         ;MULTIPLY BY 1810
ADDZL    ACY,ACY
ADDZL    ACY,ACX

```

14. Perform the inclusive OR of the operands in AC0 and AC1. The result is placed in AC1. The carry bit is unchanged.

```
COM      0,0
AND      0,1
ADC      0,1
```

15. Perform the exclusive OR of the operands in AC0 and AC1. The result is placed in AC1. The contents of AC2 and the carry bit are destroyed.

```
MOV      1,2
ANDZL    0,2
ADD      0,1
SUB      2,1
```

16. Move 30 words from locations 2000_8 - 2035_8 to locations 3000_8 - 3035_8 . The autoincrement locations are used to hold the source and destination addresses.

```

LDA      0,ADDRS      ;SET UP SOURCE ADDRESS
STA      0,20
LDA      0,ADDRD      ;SET UP DESTINATION ADDRESS
STA      0,21
LOOP:    LDA      0,@20      ;INCREMENT SOURCE ADDRESS
                ; AND GET WORD
        STA      0,@21      ;INCREMENT DESTINATION
                ; ADDRESS AND STORE WORD
        DSZ      CNT        ;DECREMENT COUNT
        JMP      LOOP      ;GO BACK FOR NEXT WORD
        ...
                ;SKIP HERE WHEN COUNT IS
                ;ZERO
        ...
ADDRS:   1777          ;SOURCE ADDRESS MINUS ONE
ADDRD:   2777          ;DESTINATION ADDRESS MINUS
                ;ONE
CNT:     36           ;WORD COUNT  $-36_8$  EQUALS  $30_{10}$ 
```

17. Perform the following unsigned integer comparisons.

```

SUB#      ACS,ACD,SZR      ;SKIP IF CONTENTS OF ACS =
                        ; CONTENTS OF ACD
SUB#      ACS,ACD,SNR      ;SKIP IF CONTENTS OF ACS =
                        ; CONTENTS OF ACD
ADCZ#     ACS,ACD,SNC      ;SKIP IF CONTENTS OF ACS <
                        ; CONTENTS OF ACD
SUBZ#     ACS,ACD,SNC      ;SKIP IF CONTENTS OF ACS <=
                        ; CONTENTS OF ACD
SUBZ#     ACS,ACD,SZC      ;SKIP IF CONTENTS OF ACS >
                        ; CONTENTS OF ACD
ADCZ#     ACS,ACD,SZC      ;SKIP IF CONTENTS OF ACS >=
                        ; CONTENTS OF ACD

```

18. Compare the signed, two's complement integer contained in ACS to 0.

```

MOV#      ACS,ACS,SZR      ;SKIP IF CONTENTS OF ACS EQ 0
MOV#      ACS,ACS,SNR      ;SKIP IF CONTENTS OF ACS NE 0
ADDO#     ACS,ACS,SBN      ;SKIP IF CONTENTS OF ACS GT 0
MOVL#     ACS,ACS,SZC      ;SKIP IF CONTENTS OF ACS GE 0
MOVL#     ACS,ACS,SNC      ;SKIP IF CONTENTS OF ACS LT 0
ADDO#     ACS,ACS,SEZ      ;SKIP IF CONTENTS OF ACS LE 0

```

19. Simulate the operation of the MULTIPLY instruction.

```

.MPYU:    SUBC 0,0          ;CLEAR AC0, DON'T DISTURB CARRY
.MPYA:    STA 3,.CB03      ;SAVE RETURN
          LDA 3,.CB20      ;GET STEP COUNT
:CB99     MOVR 1,1,SNC      ;CHECK NEXT MULTIPLIER BIT
          MOVR 0,0SKP      ;0 SHIFT
          ADDZR 2,0         ;1 - ADD MULTIPLICAND AND SHIFT
          INC 3,3,SZR      ;COUNT STEP, COMPLEMENTING CARRY ON
                        ; FINAL COUNT
          JMP .CB99        ;ITERATE LOOP
          MOVCR 1,1        ;SHIFT IN LAST LOW BIT (WHICH WAS
                        ; COMPLEMENTED BY FINAL COUNT) AND
          JMP @.CB03      ;RESTORE CARRY
.CB03:    0
.DB20:    -20              ;1610 STEPS

```

20. Simulate the operation of the DIVIDE instruction.

```
.DIVI:  SUB  0,0      ;INTEGER DIVIDE CLEAR HIGH PART
.DIVU:  STA  3,.CC03 ;SAVE RETURN
        SUB#  2,0,SZC ;TEST FOR OVERFLOW
        JMP  .CC99   ;YES, EXIT(AC0 ≥ AC2)
        LDA  3,.CC20 ;GET STEP COUNT
        MOVZL 1,1    ;SHIFT DIVIDEND LOW PART
.CC98   MOVL  0,0    ;SHIFT DIVIDEND HIGH PART
        SUB#  2,0,SZC ;DOES DIVISOR GO IN?
        SUB  2,0    ;YES
        MOVL  1,1    ;SHIFT DIVIDEND LOW PART
        INC  3,3,SZC ;COUNT STEP
        JMP  CC98    ;ITERATE LOOP
        SUB0  3,3,SKP ;DONE, CLEAR CARRY
.CC99   SUBZ  3,3    ;SET CARRY
        JMP  @.CC03 ;RETURN
.CC03   0
.CC20   20          ;1610 STEPS
```

21. Load a byte from memory. The routine is called via a JSR. The byte pointer for the requested byte is in AC2. The requested byte is returned in the right half of AC0. The left half of AC0 and the carry are set to 0. AC1 and AC2 are unchanged. AC3 is destroyed.

```
LBYT:  STA  3,LRET  ;SAVE RETURN ADDRESS
        LDA  3,MASK
        MOVR 2,2,SNC ;TURN BYTE POINTER INTO WORD ADDRESS
        ; AND SKIP IF REQUEST BYTE IS RIGHT
        ; BYTE
        MOVS 3,3    ;SWAP MASK IF REQUESTED BYTE IS LEFT
        ; BYTE
        LDA  0,0,2  ;PLACE WORD IN AC0
        AND  1,0,SNC ;MASK OFF UNWANTED BYTE AND SKIP IF
        ; SWAP IS NOT NEEDED
        MOVS 0,0    ;SWAP REQUESTED BYTE INTO RIGHT HALF
        ; OF AC0
        MOVL 2,2    ;RESTORE BYTE POINTER AND CARRY
        JMP  @ LRET ;RETURN
LRET:  0           ;RETURN LOCATION
MASK:  377
```

22. Store a byte in memory. The routine is called via a JRS. The byte to be stored is in the right half of AC0 with the left half of AC0 set to 0. The byte pointer is in AC2. The word written is returned in AC0. AC1 and AC2 are unchanged. AC3 and the carry bit are destroyed.

```

SBYT:   STA   3,SRET  ;SAVE RETURN
        STA   1,SAC1  ;SAVE AC1
        LDA   3,MASK
        MOVR  2,2,SNC ;CONVERT BYTE POINTER TO WORD
                ; ADDRESS AND SKIP IF BYTE IS TO BE
                ; RIGHT HALF
        MOVS  0,0,SKP ;SWAP BYTE AND LEAVE MASK ALONE
        MOVS  3,3     ;SWAP MASK
        LDA   1,0,2   ;LOAD WORD THAT IS TO RECEIVE BYTE
        AND   3,1     ;MASK OFF BYTE THAT IS TO RECEIVE
                ; NEW BYTE
        ADD   1,0     ;ADD MEMORY WORD ON TOP OF NEW BYTE
        STA   0,0,2   ;STORE WORD WITH NEW BYTE
        MOVL  2,2     ;RESTORE BYTE POINTER AND CARRY
        LDA   1,SAC1  ;RESTORE AC1
        JMP   @ SRET  ;RETURN
SRET:   0           ;RETURN LOCATION
SAC1:   0
MASK:   377

```

E. Instruction Execution Times

E.

INSTRUCTION MNEMONIC	RC3608	RC3609
	32K memory	16K memory
LDA	1.6 μ s	1.4 μ s
STA	1.6 μ s	1.4 μ s
ISZ, DSZ	2.4 μ s	2.1 μ s
JMP	0.8 μ s	0.7 μ s
JSR	1.25 μ s	1.2 μ s
COM, NEG, MOV, INC	1.15 μ s	1.0 μ s
ADC, SUB, ADD, AND		
Each level of @, add	0.85 μ s	0.75 μ s
Each autoindex, add	0.85 μ s	0.75 μ s
Base register addr, add	0 μ s	0 μ s
Shift R, L, add	0.3 μ s	0.3 μ s
Swap, add	0.9 μ s	0.9 μ s
If SKIP occurs, add	0.2 μ s	0.2 μ s
I/O INPUT (incl. READS, INTA)	1.85 μ s	1.81 μ s
I/O OUTPUT (MSKO)	2 μ s	2 μ s
NIO (INTEN, INTDS)	1.7 μ s	1.7 μ s
I/O SKIP	1.4 μ s	1.4 μ s
If SKIP occurs, add	0.2 μ s	0.2 μ s
For S, C and P, add	0 μ s	0 μ s
<u>DATA CHANNEL</u>		
DMA Input	1.9 μ s	1.9 μ s
DMA Output	1.8 μ s	1.8 μ s
DMA Increment	2.7 μ s	2.6 μ s
DMA Add to Memory	3.2 μ s	3.2 μ s

Executions times for extended instructions set, see E-2, E-3.

INSTRUCTION MNEMONIC

IDFY	:	1,5 μ s
LBD	RIGHT :	3,1 μ s
	LEFT :	3,7 μ s
STB	RIGHT :	4,4 μ s
	LEFT :	5,0 μ s
WMOVE	:	1,5 μ s + (number of words) x 2,7 μ s
SFREE	:	2,6 μ s + (number of occupied elements before free) x 2,3 μ s
SCHEL	:	8,7 μ s + (number of not recognized element in the link or before searched element) x (3,3 μ s - 7,0 μ s)
FETCH	:	6,7 μ s
LINK	:	7,2 μ s
REMEL	:	8,1 μ s
PLINK	:	12,6 μ s + (number of element with higher or equal priority in the link) x 2,3 μ s

TKADD	:	Modification	00	01	10	11
			4,7 μ s	4,9 μ s	4,7 μ s	7,0 μ s

TKVAL	:	Modification	00	01	10	11
			5,1 μ s	2,9 μ s	7,7 μ s	2,9 μ s

COMP : 1,2 μ s +

	BYTE 1	
BYTE 2	LEFT	RIGHT
LEFT	7,5 μ s	6,8 μ s
RIGHT	6,8 μ s	6,2 μ s

INSTRUCTION MNEMONIC

BMOVE (without convert)	:	1,5 μ s +	TO		
			FROM		
			LEFT	RIGHT	
			LEFT	7,9 μ s	7,3 μ s
			RIGHT	7,3 μ s	6,7 μ s

BMOVE (via left convert)	:	1,5 μ s +	TO		
			FROM		
			LEFT	RIGHT	
			LEFT	11,0 μ s	10,4 μ s
			RIGHT	10,4 μ s	9,8 μ s

BMOVE (via right convert)	:	1,5 μ s +	TO		
			FROM		
			LEFT	RIGHT	
			LEFT	10,4 μ s	9,8 μ s
			RIGHT	9,8 μ s	9,2 μ s

The interruptable instructions (COMP, BMOVE, WMOVE, SFREE, SCHELL and PLINK) will be elongated with 1,0 μ s if the DMA-channel interrupts or if there exist an I/O channel interrupts waiting in disable mode.

RETURN LETTER

Title: RC3803 CPU Programmer's Reference
Manual

RCSI.No.: 42-i1008

A/S Regnecentralen af 1979/RC Computer A/S maintains a continual effort to improve the quality and usefulness of its publications. To do this effectively we need user feedback, your critical evaluation of this manual.

Please comment on this manual's completeness, accuracy, organization, usability, and readability:

Do you find errors in this manual? If so, specify by page.

How can this manual be improved?

Other comments?

Name: _____ Title: _____

Company: _____

Address: _____


Date: _____

Thank you

..... **Fold here**

..... **Do not tear - Fold here and staple**

**Affix
postage
here**

 **REGNECENTRALEN**
_____ **af 1979**
Information Department
Lautrupbjerg 1
DK-2750 Ballerup
Denmark